МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет
Кафедра физико-технической информатики

И. Б. Логашенко

Практикум по методам анализа экспериментальных данных

Методическое пособие

Новосибирск
2013

Логашенко И.Б. Практикум по методам анализа экспериментальных данных. Методическое пособие / Новосиб. гос. ун-т. Новосибирск, 2013. 54 с.

В рамках курса "Методы анализа экспериментальных данных" проводятся практические занятия, на которых студенты получают опыт применения современных методов анализа результатов измерений, получаемых в физических экспериментах. В настоящем методическом пособии даны основные сведения о работе с двумя широко используемыми в физике высоких энергий пакетами программного обеспечения: пакетом ROOT для статистической обработки данных и генерации научной графики, и пакетом Geant4 для моделирования эксперимента методом Монте-Карло. В заключительной главе пособия приведены задания на различные темы, связанные с анализом данных: методы Монте-Карло, моделирование эксперимента, метод максимального правдоподобия, нейронные сети, многомерная классификация данных. Для каждого задания формулируется задача, обсуждается способ решения задачи, перечисляются дополнительные вопросы.

Автор

Логашенко Иван Борисович, канд. физ-мат. наук

Пособие подготовлено в рамках реализации Программы развития НИУ-НГУ на 2009–2018 г. г.

© Новосибирский государственный университет, 2013

Оглавление

Оглавление 3
1 Программный пакет ROOT 4
1.1 Обзор 4
1.2 Установка и запуск 4
1.3 Графики и гистограммы 6
1.4 Графические возможности 10
1.5 Функции, подгонки и минимизация 13
1.6 Хранение и доступ к данным 18
2 Программный пакет Geant4 24
2.1 Обзор 24
2.2 Алгоритм моделирования в Geant4 26
2.3 Организация программы моделирования в Geant4 28
2.4 Описание геометрии детектора 30
2.5 Описание модели взаимодействия частиц с веществом 33
2.6 Первичный генератор 34
2.7 Взаимодействие с ядром Geant4 в процессе моделирования 35
2.8 Сохранение результатов моделирования 35
3 Практические задания 36
3.1 Toy Monte-Carlo 36
3.2 Моделирование детектора 38
3.3 Метод максимального правдоподобия 41
3.4 Нейронные сети 44
3.5 Многомерная классификация данных 46
Литература 54

1 Программный пакет ROOT

1.1 Обзор

Статистический анализ данных широко применяется в многих областях знаний: физике, биологии, экономике, медицине,… Соответственно, существует множество программных пакетов для выполнения анализа данных. Большинство пакетов специализированы для определенного класса задач. Существуют и универсальные бесплатно распространяемые пакеты, например пакет R.

В физике высоких энергий (и не только) фактически стандартным пакетом для обработки данных стал программный пакет ROOT [1]. Основной причиной этого стало то, что ROOT – не только пакет обработки данных, но он также предоставляет эффективные механизмы для хранения и доступа к огромным объемам данных, характерных для современных экспериментов. Немаловажно и то, что ROOT разрабатывается широким кругом физиков, хорошо понимающих, что нужно от этого пакета конечному пользователю.

Итак, ROOT – это бесплатная, открытая, кросс-платформенная программная библиотека и интерактивная среда, предоставляющая следующие возможности:

1) инструменты для хранения и обработки больших объемов данных;
2) инструменты для статистического анализа данных, например, работа с гистограммами, подгонка экспериментальных распределений;
3) инструменты для многомерного анализа данных, например, работа с нейронными сетями;
4) инструменты для подготовки научной графики;
5) интеграцию с программами, написанными на языках C++, Python, Ruby;
6) средства для подключения внешних библиотек;
7) средства для выполнения анализа данных в пакетном режиме и в параллельных системах.

Вся информация о ROOT доступна на официальной веб-страничке [2]. Из всей документации наиболее полезны руководство пользователя [3] и детальный справочник по системе [4]. Для начинающего пользователя существует краткое практическое руководство [5]. В документации содержится большое количество примеров решения отдельных задач с помощью ROOT [6].

Пакет ROOT очень обширен и его невозможно полностью описать в рамках настоящего пособия. Здесь приводятся основные сведения, которых достаточно для первоначального знакомства с пакетом и для выполнения практических заданий. Предполагается, что читатель хорошо знаком с основами объектно-ориентированного программирования и с языком C++.

1.2 Установка и запуск

ROOT уже установлен на основном сервере терминального класса. Однако, при желании, его можно установить и на любую другую машину. На официальной веб-страничке доступны бинарные сборки пакета для систем Linux (Red Hat Enterprise version 5, 6, или Scientific Linux version 5, 6), Mac OS X, Solaris 11, Windows 7/Vista/XP, и некоторых других. Там же доступен и архив с исходными кодами и подробные инструкции по сборке и установке на различных системах.

Для запуска ROOT в системе LINUX необходимо настроить переменные окружения:

```
export ROOTSYS=…
export PATH=${PATH}:${ROOTSYS}/bin
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${ROOTSYS}/lib
```

Интерактивная среда ROOT представляет собой интерпретатор C++. Команды (в виде программы C++) можно выполнять прямо из командной строки, а можно описать их в отдельном файле и выполнить этот файл из командной строки.

Для запуска интерактивной среды используется команда root:

```
root [-l] [-b] [-n] [-q] [file1.C ... fileN.C]

Ключи:
  -b : работа в пакетном режиме без графических возможностей
  -n : не выполнять настроечные файлы, указанные в .rootrc
  -q : выйти после выполнения всех файлов, указанных в командной
       строке
  -l : не показывать экран приветствия
```

Типичный сеанс работы в ROOT выглядит следующим образом:

```
> root -l
root[0] TH1D *h = new TH1D("h1","Test histogram",100,0,10);
root[1] TF1 *f = new TF1("f1","x-x*x",0,1);
root[2] h->FillRandom("f1",1000);
root[3] h->Draw();
root[4] .q
```

В данном примере создается гистограмма, которая заполняется случайными числами согласно заданному распределению и отрисовывается. Сеанс работы представляет собой построчное выполнение программы на C++. Интерпретатор CINT, используемый в ROOT, реализует практически полный вариант языка со следующими расширениями:

1) кнопка `<TAB>` автоматически дополняет имя класса и показывает список его методов;
2) декларация типа переменных не обязательна: `c = new TCanvas();`
3) возможность использования как `.` , так и `->` для доступа к методам и членам класса: и `h->Draw()`, и `h.Draw()` можно использовать одновременно;
4) множество объектов в ROOT именованы; в этом случае возможно обращение к объекту по имени:

```
root[0] new TH1D("h1","Test histogram",10,0,10)
root[1] h1->Draw()
```

В интерпретаторе определено несколько встроенных команд, начинающихся с `.`:

```
.?              список всех дополнительных команд интерпретатора
.x [filename]   загрузить [filename] и выполнить функцию [filename]
.L [filename]   загрузить [filename]
.ls             список объектов в текущей директории
```

Доступ к различным полезным функциям интерпретатора возможен через предопределенные глобальные указатели, название которых начинается с буквы `g`:

```
gRandom - генератор случайных чисел
gFile   - указатель на текущий рабочий файл
gSystem - системная информация о текущей сессии ROOT
gROOT   - доступ к внутренней информации текущей сессии ROOT
```

Например, чтобы узнать имя машины, на которой выполняется сеанс ROOT, можно выполнить команду `gSystem->HostName()`.

Как правило, работа с ROOT организуется в полу-интерактивном режиме: пользователь пишет программу анализа (скрипт) в отдельном файле, которую потом выполняет в интерактивной сессии. Существует два основных подхода к написанию скриптов ROOT. В простых задачах удобно использовать неименованные скрипты, которые предназначены исключительно для выполнения интерпретатором CINT. В таких скриптах можно использовать все расширения языка, допустимые интерпретатором. В неименованных скриптах вся программа заключается в блок фигурных скобок. Пример такого скрипта показан на Рисунок 1а.

Более универсальным подходом является написание скрипта в виде полноценной программы на C++ (Рисунок 1б.). Преимуществом такого подхода является то, что такой скрипт можно выполнять как в интерпретаторе, так и скомпилировать его и использовать как независимую программу (в этом случае ROOT будет играть роль библиотеки классов). Более того, интерпретатор ROOT может сам скомпилировать такой скрипт с помощью системного компилятора (если при загрузке файла добавить к его названию +). Такой подход значительно более эффективен в случае больших проектов и при анализе больших массивов данных.

File script1.c:

```
{
  new TH1D("h","Example",10,0,10);
  h->Draw();
  cout << " Hello" << endl;
}
```

Использование:

```
root[0] .x script1.c
```

а) пример неименованного скрипта

File script2.c:

```
#include <iostream.h>
#include "TH1D.h"

using namespace std;

void test() {
  TH1D * h = new
      TH1D("h","Example",10,0,10);
  h->Draw();
  cout << "Hello" << endl;
}
```

Использование:

```
root[0] .L script2.c
или
root[0] .L script2.c+
root[1] test()
```

б) пример полного скрипта

Рисунок 1. Пример исполняемых скриптов в среде ROOT

1.3 Графики и гистограммы

Основными средствами представления данных в пакете ROOT являются графики (`TGraph…`) и гистограммы (`TH…`).

Графики

График (graph) – это коллекция точек на плоскости (в пространстве). Существуют следующие основные типы графиков (приведен один из возможных конструкторов):

- `TGraph(n,x,y)` – коллекция точек с координатами `x[]` и `y[]`; `n` задает размерность массивов `x` и `y`.

- `TGraphErrors(n,x,y,ex,ey)` – коллекция точек с координатами `x[]` и `y[]`; n задает размерность массивов `x` и `y`. Для каждой точки задается величина ошибки вдоль горизонтальной (`ex`) и вертикальной (`ey`) осей.
- `TGraphAsymmErrors(n,x,y,exl,exh,eyl,eyh)` – коллекция точек с координатами `x[]` и `y[]`; n задает размерность массивов `x` и `y`. Для каждой точки задается величина ошибок с обеих сторон от центрального значения вдоль горизонтальной (`exl, exh`) и вертикальной (`eyl, eyh`) осей.
- `TGraph2D(n,x,y,z)` – коллекция точек в пространстве с координатами `x[], y[]` и `z[]`; n задает размерность массивов `x,y` и `z`.

Как правило, массивы с координатами точек заполняются до создания графика. Можно создавать пустой график с заданным количеством точек и затем заполнять его поточечно с помощью метола `g->SetPoint(i,value)`.

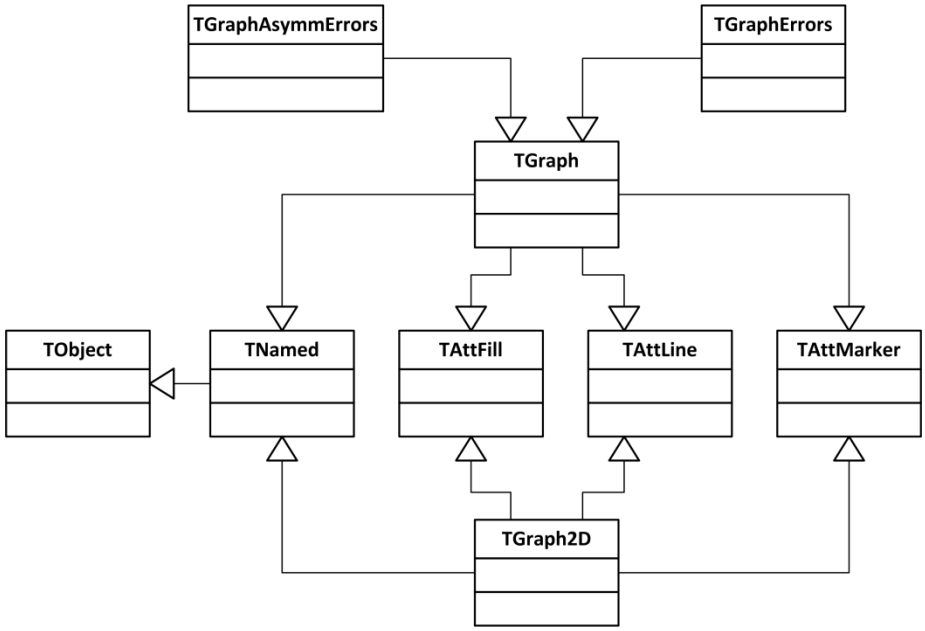


Рисунок 2. Диаграмма классов TGraph...

На Рисунок 2 показана диаграмма классов, связанных с графиками. Отметим следующие характерные особенности структуры классов в пакете ROOT.

1) Все классы в пакете, которые представляют собой хранимый объект, наследуют от единого базового класса `TObject`. Базовый класс задает интерфейс (и, частично, реализацию) работы с объектами для изучения внутренней структуры объектов, отрисовки, сортировки, организации коллекций объектов, превращения объектов в байтовую строку и т.п.
2) Основные классы, которые представляют собой уникальные хранимые объекты, наследуют от базового класса `TNamed`. Все объекты, наследованные от `TNamed`, идентифицируются уникальным символьным идентификатором. Такой подход позволяет осуществлять поиск таких объектов в памяти и в файле.
3) Объекты, которые можно отрисовать, наследуют от специальных классов, в которых указаны параметры рисования. В данном случае, такими классами являются: `TAttLine` – параметры отрисовки линий; `TAttMarker` - параметры отрисовки са-

мих точек, принадлежащих графику; `TAttFill` – параметры закрашивания сплошных областей.

Пример создания двух графов:

```
const int n = 5;
double x[n] = { 1, 2, 3, 4, 5};
double y[n] = { 0.1, 1.0, 2.0, 5.0, 10.0 };
double ey[n] = { 0.1, 0.1, 0.1, 0.1, 0.1 };
TGraph *g1 = new TGraph(n,x,y);
TGraphErrors *g2 = new TGraphErrors(n,x,y,0,ey);
```

Гистограммы

Гистограмма (histogram) – другой очень распространенный способ описания данных. В простейшем случае, для построения гистограммы диапазон изменения некоторой величины разбивается на `n` одинаковых интервалов (бинов), и подсчитывается число случаев (событий), когда значение данной величины попало в каждый интервал. Существуют более сложные варианты гистограмм: для каждого бина может сохраняться не число событий, а среднее значение; диапазон изменения может быть многомерным; интервалы могут быть разного размера, и т.п.

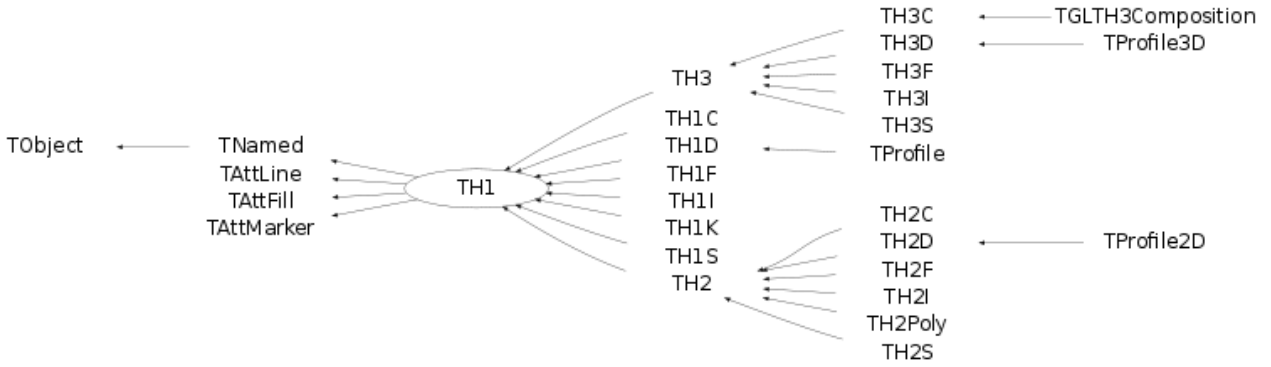
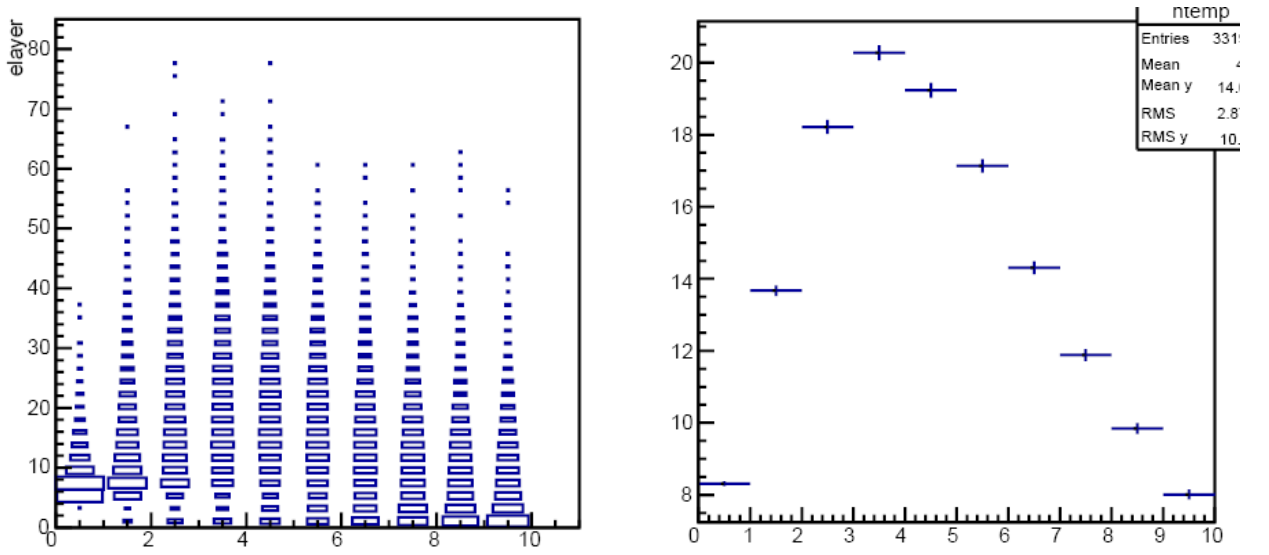


Рисунок 3. Диаграмма классов гистограмм

В пакете ROOT существует целое семейство классов для описания гистограмм. Диаграмма классов изображена на Рисунок 3. Имя класса определяется размерностью гистограммы и ее содержимым: `TH{D}{T}`, где `D` – задает размерность, а `T` – тип хранимых данных. Одномерные гистограммы (`D=1`) представляют собой отрезок, разбитый на интервалы; двумерные гистограммы (`D=2`) представляют собой прямоугольник, разбитый на ячейки; наконец, трехмерные гистограммы (`D=3`) представляют собой прямоугольный параллелепипед. В каждой ячейке гистограммы хранится один элемент данных определенного типа: однобайтовое целое, char (`T=C`); двухбайтовое целое, short (`T=S`); четырехбайтовое целое, int (`T=I`); вещественное, float (`T=F`); вещественное двойной точности, double (`T=D`). Например:

- TH1I хорошо подходит для определения классической гистограммы, в которой хранится число событий, попавших в каждый интервал, если это число не превышает 2 147 483 647.
- TH2C хорошо подходит для хранения одного цветового канала для RGB изображения (максимальное значение 255).
- TH1D хорошо подходит для хранения взвешенной суммы, вычисленной по всем событиям, попавшим в каждый интервал, в случае, когда сумма может принимать любые значения.

Особую роль играют «профильные» гистограммы `TProfile` (`TProfile2D`, `TProfile3D`). В них для каждой ячейки хранится не одно число (как правило, количество событий), а число с ошибкой. Например, `TProfile` хорошо подходит для хранения средней зарплаты и разброса (дисперсии) зарплат для разных возрастных категорий сотрудников. Заметим, что в простых гистограммах со значением для каждой ячейки также ассоциируется величина ошибки. Разница между простой гистограммой и профильной заключается в способе вычисления самого значения и его ошибки. В простой гистограмме значение – это сумма весов всех событий, попавших в интервал, а ошибка по умолчанию вычисляется как корень из этой суммы. В профильной гистограммы значение – это средняя величина по всем событиям, попавшим в интервал, а ошибка вычисляется как выборочная дисперсия соответствующего распределения. На Рисунок 4 показано представление одних и тех же данных в виде двумерной гистограммы и в виде профильной гистограммы.



TH2D("h2","2D histogram", 11,0,11,100,0,100)

TProfile("hp","Profile histogram", 11,0,11,0,100)

Рисунок 4. Представление данных в виде двумерной и профильной гистограмм

Пример создания и заполнения гистограмм:

```
TH1I *h1 = new TH1I("h1","1D hist",100,-5,5);
TH2D *h2 = new TH2D("h2","2D hist",40,-10,10,40,-10,10);
TProfile *hp = new TProfile("hp","Profile",40,-10,10);
for( int i=0; i<500; i++ ) {
   double r = gRandom->Gaus(0,1);
   double x = gRandom->Uniform(-10,10);
   double y = x+r;
   h1->Fill(r);
   h2->Fill(x,y);
   hp->Fill(x,y);
}
```

Получившиеся гистограммы показаны на Рисунок 5.

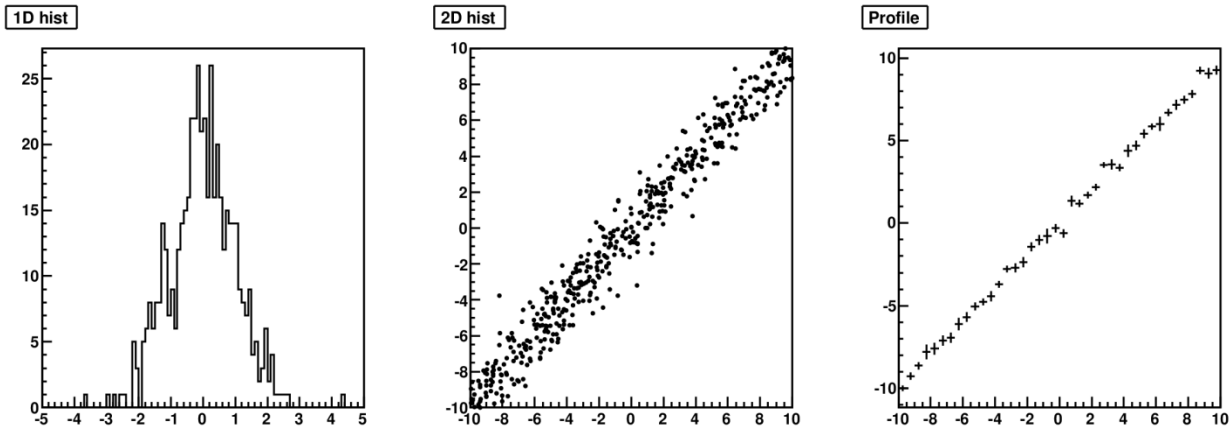


Рисунок 5. Примеры одномерной, двумерной и профильной гистограмм

1.4 Графические возможности

Важнейшим элементом анализа данных является их графическое представление. В пакете ROOT содержится множество гибких механизмов отрисовки данных. Для создания картинок нужно сначала создать поле для рисования (`TCanvas`, `TPad`) и затем добавлять на него объекты с помощью методов соответствующих классов.

Для добавления нового поля для рисования необходимо создать объект класса `TCanvas`, для которого задается идентификатор, заголовок и размеры в пикселях по горизонтали и вертикали:

```
TCanvas *c = new TCanvas("c","My Canvas",800,600)
```

При необходимости отрисовки нескольких объектов на одной картинке, поле для рисования можно разбить на несколько независимых ячеек (`TPad`). Класс `TCanvas` предоставляет удобный табличный способ расположения объектов с помощью метода `Divide(nx,ny)`, где `nx` и `ny` – число столбцов и строк, соответственно. Для более сложного расположения объектов необходимо создавать ячейки «вручную» с помощью конструктора `TPad`. Пример использование табличного расположения показан на Рисунок 5, который был сгенерирован следующим образом:

```
TCanvas *c = new TCanvas("c","My Canvas",1000,500);
c->Divide(3,1);
c->cd(1);
h1->Draw();
c->cd(2);
h2->Draw();
c->cd(3);
hp->Draw();
```

Собственно отрисовка графиков и гистограмм (и других объектов) производится с помощью метода `Draw()` соответствующего класса. Параметры отрисовки задаются либо как аргументы этого метода, либо с помощью методов определенных базовых классов. Большинство объектов, которые можно нарисовать, наследуют следующие методы.

- Из класса `TAttMarker` наследуются методы `SetMarkerStyle(istyle)`, `SetMarkerColor(icolor)`, `SetMarkerSize(float)`. Маркер обозначает каждую точку (бин) графика (гистограммы). На Рисунок 6 показаны все основные

доступные маркеры и соответствующие значения `istyle`. Таблица основных цветов, используемых при рисовании в ROOT, показана на Рисунок 7.

- Из класса `TAttLine` наследуются методы `SetLineStyle(istyle)`, `SetLineColor(icolor)`, `SetLineWidth(int)`. Эти методы позволяют задать основные параметры линий, используемые при отрисовке объекта (например, рисовать гистограмму сплошной или прерывистой линией). На Рисунок 8 показаны все основные доступные типы линий и соответствующие значения `istyle`.
- Из класса `TAttFill` наследуются методы `SetFillStyle(istyle)`, `SetFillColor(icolor)`. Эти методы позволяют задать основные параметры закрашивания, используемые при отрисовке объекта (например, рисовать гистограмму заштрихованной, или заполненной заданным цветом).

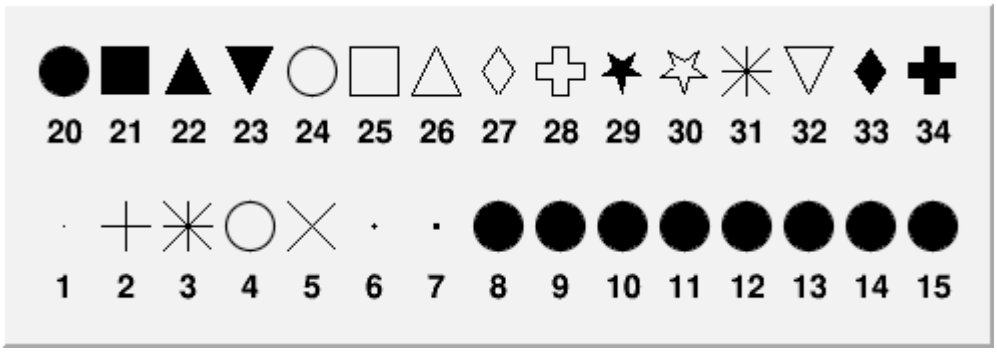


Рисунок 6. Таблица маркеров, используемых при отрисовке графиков и гистограмм.

Рисунок 7. Таблица цветов, используемых при отрисовке графиков и гистограмм

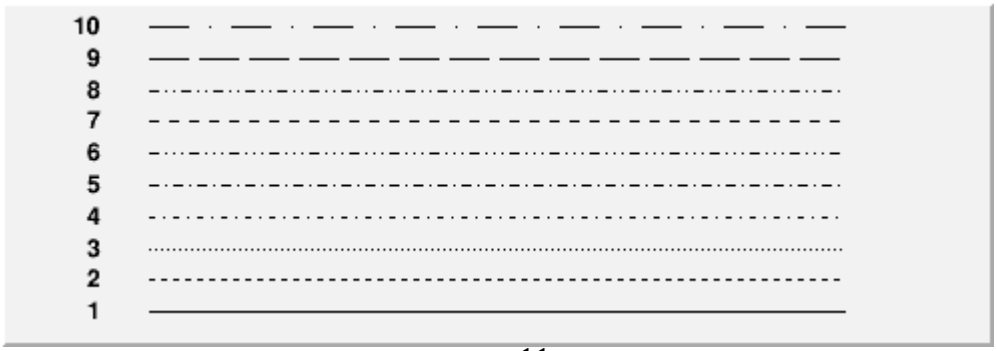


Рисунок 8. Типы линий, используемых при отрисовке графиков и гистограмм

Отрисовка графиков производится с помощью метода `TGraph::Draw(char *)`. Наиболее часто используемые значения аргумента метода `Draw()`:

- “`L`” - точки на графике соединяются линиями;
- “`A`” – рисуются оси с автоматическим выбором масштаба;
- “`P`” – каждая точка отмечается маркером.

Аргументы метода `Draw()` можно объединять, например: `Draw(“AP”)`. На рисунке Рисунок 9 показан пример отрисовки нескольких графиков.

```
new TCanvas(“c”,” “,800,800);

// Красная линия
g1 = new TGraph(n,x,y);
g1->SetLineColor(2);
g1->SetLineWidth(3);
g1->Draw("AL");

// Синие точки с ошибками
g2 = new TGraphErrors(n,x1,y1,0,ey1);
g2->SetLineColor(4);
g2->SetMarkerColor(4);
g2->SetMarkerStyle(20);
g2->Draw("P");
```

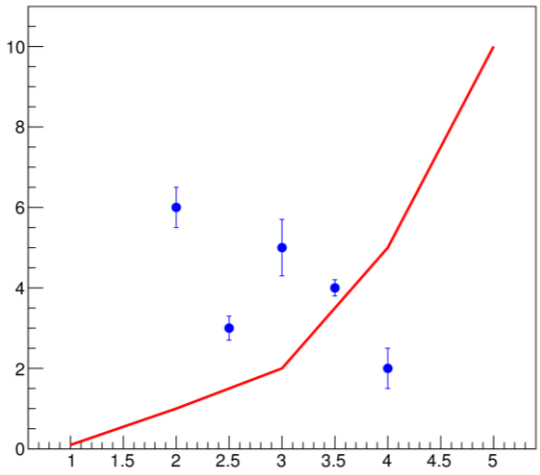


Рисунок 9. Пример отрисовки двух графиков

Отрисовка гистограмм производится с помощью метода `TGraph::Draw(char *)`. Наиболее часто используемые значения аргумента метода `Draw()` для одномерных гистограмм:

- “`E`” – рисовать гистограмму как точки с ошибками;
- “`L`” – рисовать линию, соединяющую точки или бины;
- “`P`” – рисовать маркер для каждого бина;

Наиболее часто используемые значения аргумента метода `Draw()` для двумерных гистограмм:

- “`BOX`” – отображать каждую ячейку в виде прямоугольника, площадь которого пропорциональна числу событий (содержимому) ячейки;
- “`LEGO`” – отображать гистограмму в виде вертикальных столбиков;
- “`SURF`” – отображать гистограмму в виде поверхности;
- “`SCAT`” – отображать гистограмму как облако разбросанных точек, причем локальная плотность точек пропорциональна числу событий в ячейке.

Если использовать аргумент “`SAME`”, то гистограмма будет нарисована в текущем окне вместе с ранее отрисованными гистограммами (без этого аргумента текущее окно будет очищено перед отрисовкой гистограммы). Аргументы метода `Draw()` можно объединять, например: `Draw(“P SAME”)`. Все возможные способы управления графическим представлением гистограмм подробно описаны в документации к пакету ROOT. На Рисунок 10 показан пример отрисовки нескольких гистограмм.

```
TCanvas *c1 = new TCanvas();

c1->Divide(1,2);

c1->cd(1);
h2->Draw("LEGO1");

c1->cd(2);
h2->Draw("BOX");

hp->SetMarkerColor(2);
hp->SetMarkerStyle(20);
hp->SetLineColor(2);
hp->SetLineWidth(3);
hp->Draw("SAME");
```

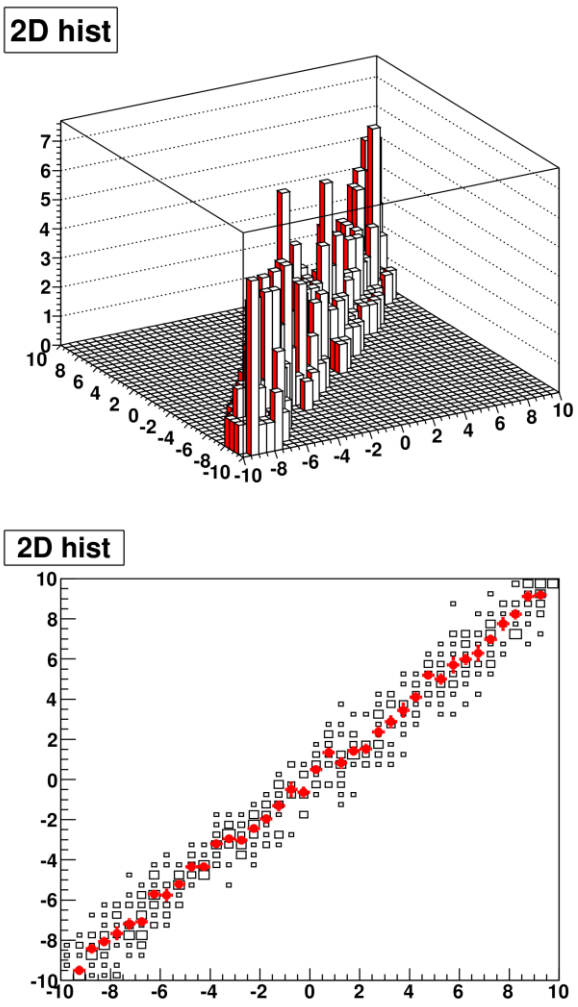


Рисунок 10. Пример отрисовки гистограмм

Помимо графиков и гистограмм, в пакете ROOT определено множество других классов, представляющие различные графические объекты. Как правило, для отрисовки таких объектов нужно использовать метод `Draw()`. Ниже перечислены наиболее полезные из них.

- `TLine`, `TPolyLine`, `TArrow` – позволяют нарисовать линию, ломанную или стрелку;
- `TEllipse` – позволяет нарисовать окружность, эллипс и дугу;
- `TBox` – позволяет нарисовать прямоугольник;
- `TText`, `TLatex` – позволяют делать надписи на рисунках; при использовании `TLatex` можно генерировать сложные математические выражения;
- `TAxis` – позволяет нарисовать координатные оси.

1.5 Функции, подгонки и минимизация

Одна из наиболее востребованных операций при анализе данных – подгонка экспериментальных данных заданной кривой (моделью) и оценка значений параметров модели. В пакете ROOT присутствует множество инструментов для решения этой задачи.

Для того, чтобы реализовать подгонку, необходимо каким то образом определить модель, которая будет использована для описания данных. Чаще всего модель описывается в виде функции, у которой есть несколько параметров, значения которых требуется определить из подгонки. Для определения таких функций в пакете ROOT существует семейство классов `TF1`, `TF2`, `TF3` (индекс соответствует размерности аргумента функции).

Определение функции с помощью встроенного интерпретатора

Функцию можно определить несколькими способами. Самый простой способ – использовать встроенный интерпретатор формул `TFormula`. Создание объекта-функции $f(x) = e^{-x}\sin x$ в этом случае выглядит следующим образом:

```
TF1 *f = new TF1("myfun","sin(x)*exp(-x)",0,10);
```

Первый аргумент конструктора, `"myfun"` – это уникальное символьное имя объекта, с помощью которого с этим объектом могут работать другие классы. Последние два аргумента конструктора, `"0,10"`, указывают область определения функции. В качестве аргументов функции используются: `x` для `TF1`; `x,y` для `TF2`; `x,y,z` для `TF3`. Если у функции есть параметры, они записываются как `[i]`, где `i` – порядковый номер параметра. Например, создать двумерную функцию $f(x,y;\omega,\lambda) = e^{-\lambda x}\sin \omega y$ с параметрами λ и ω можно следующим образом:

```
TF2 *f = new TF2("myfun","exp(-[0]*x)*sin([1]*y)",0,10,0,10);
```

Здесь первым параметром `[0]` является λ, а вторым параметром `[1]` – ω.

В символьном выражении функции можно использовать стандартные математические функции (`sin`, `cos`, `exp`, `log`,…), математические функции, определенные в классе `TMath` (`TMath::Erf(x)`, `TMath::Landau(x)`,…), арифметические операции, операцию возведения в степень (`**`, `^`), логические операции (`&&`, `||`, `==`, `<=`, `>=`, `!`). Примеры допустимых символьных выражений:

```
sin(x)/x
[0]*sin(x) + [1]*exp(-[2]*x)
x + y**2
x^2 + y^2
[0]*pow([1],4)
2*pi*sqrt(x/y)
```

Дополнительно к перечисленным, существует несколько предопределенных функций с параметрами, часто используемых на практике:

- `gaus` – ненормированный гауссиан `[0]*exp(-0.5*((x-[1])/[2])**2)`
- `gausn` – нормированный гауссиан
  `[0]*exp(-0.5*((x-[1])/[2])**2)/(sqrt(2*pi)*[2]))`
- `expo` – экспоненциальная зависимость `exp([0]+[1]*x)`
- `polN` – полином `N`-той степени, например
  `pol3 = par[0]+par[1]*x+par[2]*x**2+par[3]*x**3`

Например, следующая функция с 5 параметрами $f(x) = a + bx + \frac{c}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-x_0)^2}{2\sigma^2}\right)$ может быть определена с помощью выражения `pol1(0)+gausn(2)`. Число в скобках указывает на порядковый номер первого параметра для данной функции:

```
gausn(2)= [2]*exp(-0.5*((x-[3])/[4])**2)/(sqrt(2*pi)*[4]))
```

Определение функции с помощью пользовательской программы

В случае, если возможностей интерпретатора формул недостаточно, можно определить объект TF1 (TF2, TF3) с помощью пользовательской функции на языке C++. Интерфейс пользовательской функции должен иметь вид:

```
Double_t myfunc(Double_t *x,Double_t *par)
```

где x – указатель на массив аргументов (x[0] – это x, x[1] – это y, и т.п.), а par – указатель на массив параметров (par[0] – первый параметр, par[1] – второй, и т.п.). Указатель на такую функцию просто передается конструктору TF1, при этом кроме области определения нужно указать еще и число параметров функции. Ниже показан пример определения функции $f(x;\omega,\lambda) = e^{-\lambda x}\sin\omega x$ с параметрами λ и ω с помощью пользовательской программы.

```
// пользовательская реализация функции на C++
Double_t myfunc(Double_t *x,Double_t *par) {
  double x1=x[0];
  double lambda=par[0], omega=par[1];
  double fval = exp(-lambda*x1)*sin(omega*x1);
  return fval;
}

// скрипт ROOT, в котором используется эта функция
void test() {
  TF1 *f1 = new TF1("f1",myfunc,0,10,2); // 2 – число параметров
}
```

Работа с функциями

Для доступа к параметрам функции можно использовать методы:

- SetParameter(i,value) – установить значение i-того параметра равным value.
- SetParameters(val1,val2,…) – установить значение первого параметра равным val1, второго – val2, и т.д.
- GetParameter(i) – вернуть значение i-того параметра.
- GetParameters(double *pars) – массив pars будет заполнен значениями параметров.

Функцию можно нарисовать с помощью метода Draw(), аналогично тому, как отрисовываются графики и гистограммы.

Подгонка гистограмм функцией

Для подгонки гистограммы функцией используется метод Fit() объекта-гистограммы:

```
void Fit(TF1 *f, Option_t *option, Option_t *goption,
         Axis_t xxmin, Axis_t  xxmax)
```

где f – указатель на функцию, option – список параметров, управляющих подгонкой, goption – список параметров, управляющих отображением функции, xxmin,xxmax – интервал области определения функции, в котором ведется подгонка. Все атрибуты, кроме f, необязательны. Вместо указателя на функцию можно использовать символьное имя одной из предопределенных функций gaus, gausn, expo, polN, или их комбинацию, например "pol1(0)+gausn(2)".

Для подгонки гистограммы по умолчанию используется метод наименьших квадратов, т.е. вычисляется сумма χ^2 и подбираются такие значения параметров функции, что χ^2 минимальна:

$$\chi^2 = \sum_{j=1}^{nbin} \frac{\left(y_j - f(x_j)\right)^2}{\sigma_j^2}$$

где x_j – середина -того бина гистограммы, y_j – значение в этом бине, а σ_j – ассоциированная ошибка. Для того, чтобы использовать метод максимального правдоподобия, в котором минимизируется логарифмическая функция правдоподобия $-\ln L$:

$$-\ln L = -\sum_{j=1}^{nbin} n_j \ln f(x_j)$$

где n_j – число событий в -том бине гистограммы, необходимо использовать `option="L"`. По умолчанию, при построении χ^2 или $-\ln L$ используется значение функции в середине каждого бина $f(x_j)$. Более правильно, но и более ресурсоемко, вычислять среднее значение функции в каждом бине:

$$f(x_j) \to \frac{1}{b_j - a_j} \int_{a_j}^{b_j} f(x')dx'$$

где a_j и b_j – границы -того бина гистограммы. Для этого нужно использовать `option="I"`.

По умолчанию, при поиске минимума χ^2 или $-\ln L$ все параметры функции могут варьироваться без ограничений. Ограничить диапазон возможных значений `i`-того параметра можно с помощью метода `TF1::SetParLimits(i,vmin,vmax)`, где `(vmin,vmax)` – допустимый интервал значений. Иногда необходимо зафиксировать значение определенного параметра и не менять его при минимизации. Это можно сделать с помощью метода `TF1::FixParameter(i,value)`, где `i` – номер параметра.

После выполнения процедуры минимизации оптимальные значения параметров функции можно извлечь с помощью метода `TF1::GetParameter(i)` (или `TF1::GetParameters()`), а оценку ошибки (дисперсии) параметров – с помощью `TF1::GetParError(i)`, где `i` – номер параметра. Значение χ^2 при оптимальном значении параметров можно получить с помощью метода `TF1::GetChisquare()`.

Для подгонки графиков используется метод `TGraph::Fit()`, аналогичный `TH::Fit()`.

На Рисунок 11 показан пример подгонки гистограммы суммой нормального распределения и линейной функции.

Минимизация

Метод `Fit()` значительно упрощает процедуру подгонки и оценки параметров с помощью методов максимального правдоподобия и наименьших квадратов в стандартных случаях. Его преимущество состоит в том, что не нужно явно записывать вид χ^2 или $-\ln L$, эти суммы неявно вычисляются внутри метода. Однако в менее стандартных случаях может оказаться, что необходимо модифицировать минимизируемую функцию – например, при использовании небинированной функции правдоподобия. В этих случаях необходимо задать соответствующую функцию в явной форме и численно найти положение ее минимума.

```
// Определим функцию
TF1 *f = new TF1("myf",
         "pol1(0)+gausn(2)",0,10);
f->SetParameters(0,0,100,5,1);
f->SetLineColor(2);

// Подгонка методом макс.правд.
h->Fit(f,"LI");
```

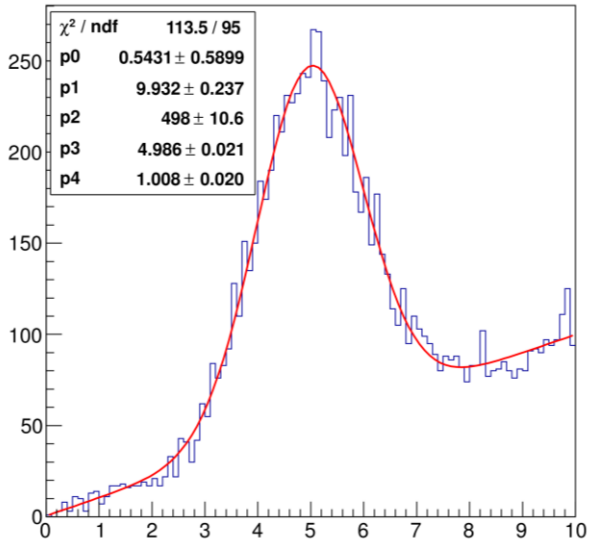


Рисунок 11. Пример подгонки гистограммы

В пакете ROOT есть несколько классов, которые позволяют находить минимум функции, заданной пользователем. Здесь мы рассмотрим только один из таких классов, `TMinuit`. Этот класс (а в исходной реализации – библиотека на языке FORTRAN) широко используется при анализе данных в физике высоких энергий уже несколько десятилетий.

Продемонстрируем принципы использования класса `TMinuit` на примере. Пусть требуется подогнать экспериментальные данные прямой линией, однако при этом известно, что в данных есть «выбросы». В этом случае необходимо использовать модифицированный метод наименьших квадратов, в котором игнорируется вклад далеко лежащих измерений. Пример минимизации подобной модифицированной функции χ^2 показан на Рисунок 12, а реализация этого примера с помощью класса `TMinuit` на Рисунок 13. В данном примере реализован один из вариантов робастной подгонки линейной функцией, в котором ограничивается максимальное локальное значение невязки тремя стандартными отклонениями:

$$\chi^2 = \sum_j \min\left[\frac{\left(y_j - f(x_j)\right)^2}{\sigma_j^2}, 9\right], \quad f(x) = ax + b$$

Черная линия показывает результаты оптимизации при использовании обычного метода наименьших квадратов, с помощью метода `h->Fit("pol1")`, а красная линия – результаты описанной робастной оптимизации. Видно, что черная линия «притянута» к выброшенным точкам, в то время, как красная их «игнорирует».

Пояснения к примеру:

1) Минимизируемая функция. Здесь это сумма хи-квадрат, в которой, в случае, если разница между измеренным и ожидаемым значением превышает 3 стандартных отклонения, разница фиксируется на этом значении.
2) Подготовка данных. Экспериментальные данные копируются из гистограммы в локальные масиивы, которые используются для вычисления минимизируемой функции.

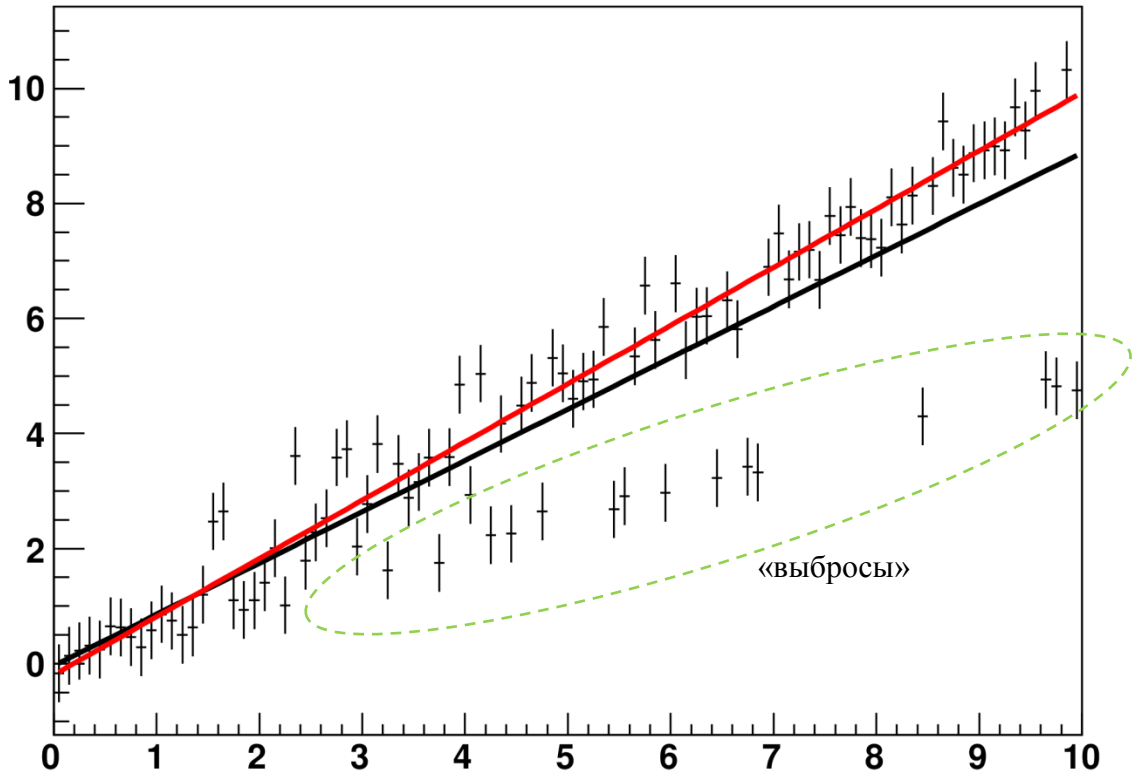


Рисунок 12. Пример реализации робастной подгонки данных с помощью класса TMinuit

3) Инициализация объекта класса `TMinuit`. В конструкторе указывается максимальное число параметров, по которым будет проводится поиск оптимума (здесь – `2`). Для передачи параметров используется устаревший подход, при котором значения параметров указываются в массиве (здесь `arglist`).
4) Определения и инициализация всех параметров, по которым будет проводиться поиск минимума (здесь – a и b).
5) Собственно, поиск минимума. При запуске указывается максимальное число итераций (здесь – `500`).
6) Чтение результатов.

1.6 Хранение и доступ к данным

Пакет ROOT предоставляет мощный механизм хранения экспериментальных данных – *деревья* (*trees*). Деревья представляют собой электронную таблицу, которая предоставляет пользователям:

1) возможность хранения практически неограниченного числа атрибутов для каждого события;
2) возможность хранения как простых аргументов (например, число), так и сложных (массив, вектор, объект);
3) возможность хранения практически неограниченного числа событий;
4) возможность хранения данных во многих файлах;
5) прозрачный доступ к данным;
6) возможность параллельной обработки данных в многопроцессорных системах.

```
const int nbin = 100;
double x[nbin],y[nbin],ey[nbin];

// минимизируемая функция
void fcn(int &npar, double *gin, double &f, double *par, int iflag)
{
   // вычисляем модифицированную сумму хи-квадрат
   double chisq = 0;
   for(int i=0;i<nbins; i++) {
     double delta  = pow((y[i]-par[0]*x[i]-par[1])/ey[i],2);
     if( delta>9 ) delta=9;
     chisq += delta;
   }
   f = chisq;
}

// основная функция
void test() {

   for( int i=1; i<=nbin; i++ ) {
      x[i] = h->GetBinLowEdge(i)+h->GetBinWidth(i)/2;
      y[i] = h->GetBinContent(i);
      ey[i] = h->GetBinError(i);
   }

   TMinuit *gMinuit = new TMinuit(2);
   gMinuit->SetFCN(fcn);

   Double_t arglist[10];
   Int_t ierflg = 0;
   arglist[0] = 1;
   gMinuit->mnexcm("SET ERR", arglist ,1,ierflg);

   Double_t vstart[2] = {1, 0};
   Double_t step[2] = {0.1 , 0.1};
   gMinuit->mnparm(0, "a", vstart[0], step[0], 0,0,ierflg);
   gMinuit->mnparm(1, "b", vstart[1], step[1], 0,0,ierflg);

   arglist[0] = 500;
   arglist[1] = 1.;
   gMinuit->mnexcm("MIGRAD", arglist ,2,ierflg);

   double a, aerr, b, berr;
   gMinuit->GetParameter(0,a,aerr);
   gMinuit->GetParameter(1,b,berr);

}
```

1 **2** **3** **4** **5** **6**

Рисунок 13. Пример использования класса TMinuit

Как правило, в современных экспериментах в физике высоких энергий, деревья ROOT являются основным механизмом хранения данных эксперимента. Существует несколько

способов работы с деревьями. Здесь мы рассмотрим самый простой, в котором в дереве сохраняются только простые аргументы.

Создание и заполнение дерева

Для создания дерева, нужно открыть файл формата ROOT с помощью создания объекта класса `TFile` и создать дерево, создав объект класса `TTree`. У дерева двухуровневая структура: дерево состоит из *ветвей* (`TBranch`), в которых хранятся *листья*. В простом подходе для каждого атрибута события создается отдельная ветвь, в которой хранится только один лист – значение атрибута. Для создания ветви используется метод `TTree::Branch()`.

В следующем примере для каждого события сохраняется три атрибута: целочисленный тип частицы (`pid`), энерговыделение в калориметре (`energy`) и массив из трех компонент импульса (`p`).

```
int pid;
float energy;
float p[3];
TFile *f = new TFile("tree.root","recreate");
TTree *t = new TTree("myt","my tree");
t->Branch("pid",    &pid,    "pid/I")
t->Branch("energy",&energy,"energy/F");
t->Branch("p",       p,      "p[3]/F");
```

Для каждой ветви указывается ее имя, указатель на область памяти, где лежат значения листьев (эта информация нужна для организации доступа к данным), и перечисляются идентификаторы листьев и их тип. Например, `pid/I` означает, что имя листа `pid`, а его тип - целочисленный. Обратите внимание, что в случае массива не используется операция создания указателя `&`, т.к. идентификатор массива уже является указателем.

Для заполнения дерева нужно просто записать требуемые значения в соответствующие области памяти и вызвать метод `TTree::Fill()`:

```
pid = 2;
energy = 231.3;
p[0]=0; p[1]=-1.2; p[2]=2.0;
t->Fill();
```

Как правило, эти операции выполняются в цикле. Например, перед началом набора данных открывается файл и создается дерево. Во время набора данных при возникновении очередного события заполняются значения всех аргументов, после чего вызывается метод `Fill()`. Эта процедура производится многократно, при этом в дерево записывается множество независимых событий, которые обладают единой структурой.

По концу записи необходимо сохранить дерево в файле и закрыть файл:

```
t->Write();
f->Close();
```

Работа с деревом

Если при запуске интерактивной оболочки ROOT указать в командной строке имя файла, он будет автоматически открыт. ROOT автоматически создает указатели на объекты, хранящиеся в файлах формата ROOT. Поэтому следующая последовательность команд покажет информацию о дереве:

```
> root -l tree.root
root[0] myt->Print()
```

Основные команды среды ROOT для работы с деревом в интерактивном режиме:

- `myt->Print()` – вывод информации о структуре дерева, числе событий и т.д.
- `myt->Show(i)` – распечатать `i`-тое событие.
- `myt->Scan("pid:energy")` – распечатать таблицу значений перечисленных аргументов для всех событий.

Наиболее мощный способ работы с данными предоставляет метод TTree::Draw():

```
Draw(char* varexp, char* selection = "", char* option = "")
```

Первый аргумент метода содержит список того, что надо нарисовать, второй аргумент – условия отбора интересных событий, третий аргумент – параметры отрисовки. Как при указании того, что необходимо нарисовать, так и при задании критериев отбора, можно использовать любые функции от аргументов, сохраненных в дереве.

Если требуется нарисовать несколько независимых параметров, то они разделяются в списке двоеточием. По умолчанию, если первый аргумент содержит только один параметр, то результат представляется в виде гистограммы, отражающей распределение заданного параметра. Если требуется отрисовать 2 или 3 независимых параметра, то результат представляется в виде двумерного или трехмерного облака точек.

Во втором аргументе метода Draw() можно указать критерии отбора. Как правило, критерии отбора – это логическая функция, записанная на языке C, например `"pid==1"`. Только те события, для которых значение этой функции истинно, попадут в отображаемую гистограмму (или облако точек).

В третьем аргументе метода Draw() указываются параметры отрисовки. В основном, список возможных значений этого аргумента совпадает с аналогичным списком для методов `TH::Draw()` и `TGraph::Draw()`.

На Рисунок 14 приведено несколько примеров использования `Draw()`. Слева направо, сверху вниз:

- `myt->Draw("energy")` – распределение энерговыделения (значений `energy`) для всех событий в дереве;
- `myt->Draw("energy","pid==2")` – распределение энерговыделения (`energy`) для событий второго типа (`pid==2`);
- `myt->Draw("energy:pid")` – распределение энерговыделения (`energy`) в зависимости от типа частицы; это двумерное распределение отображено в виде прямоугольных ячеек, площадь которых пропорциональна числу событий в ячейке;
- `myt->Draw("energy:pid","","prof")` – распределение среднего энерговыделения (`energy`) в зависимости от типа частицы; на этот раз вместо двумерного распределения сформирована профильная гистограмма, поскольку задана опция отрисовки “`prof`”;
- `myt->Draw("sqrt(p[0]**2+p[1]**2)")` – распределение значения поперечного импульса $\sqrt{p_x^2 + p_y^2}$;
- `myt->Draw("p:Iteration$","p[2]>0")` – распределение значений компонент импульса для событий с $p_z > 0$. Конструкция `Iteration$` означает неявный цикл по элементам массива. Например, в данном случае, для отобранных событий в двумерное распределение будут добавлены точки (`p[0],0`), (`p[1],1`), (`p[2],2`).

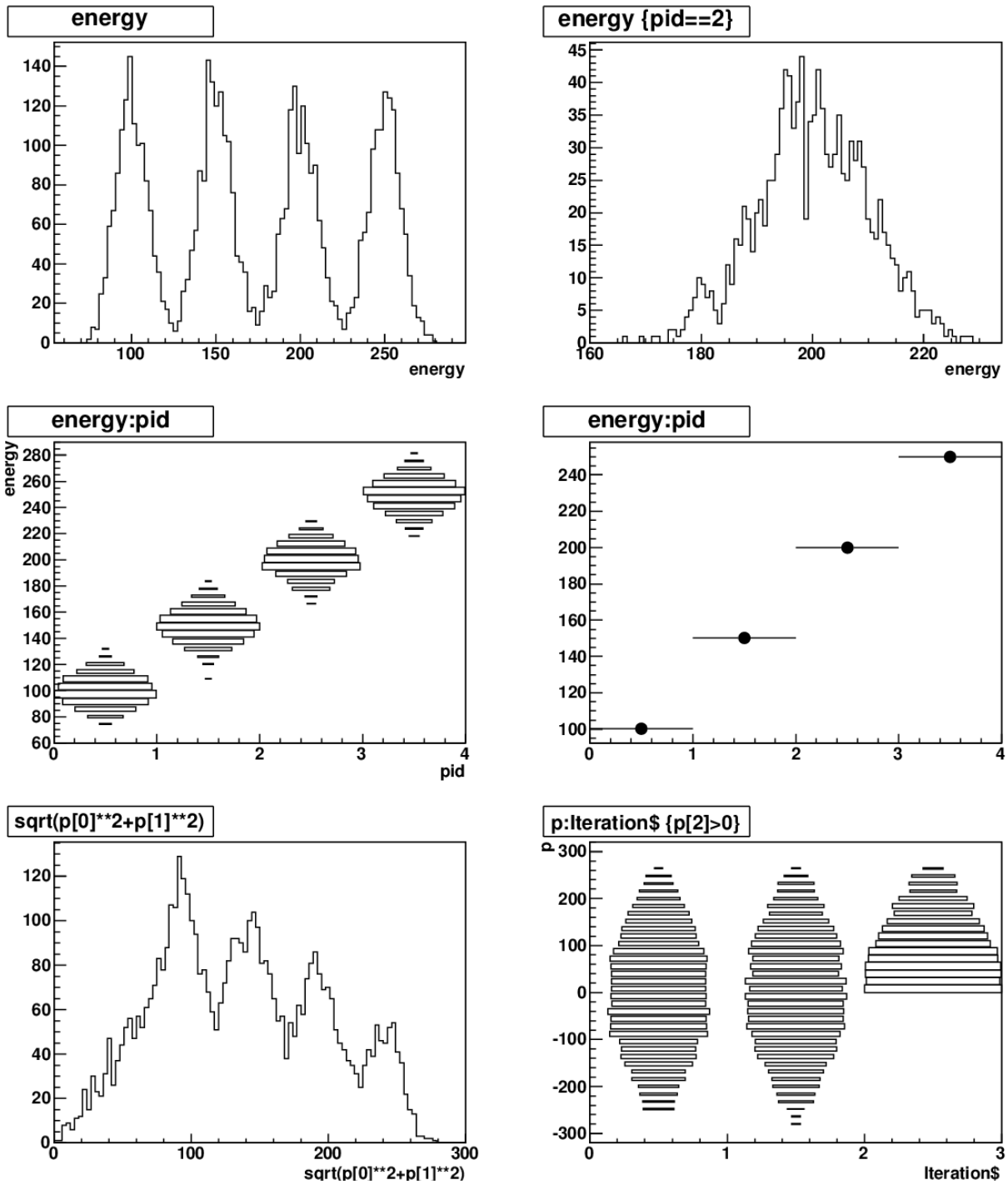


Рисунок 14. Примеры использования метода TTree::Draw()

Метод `Draw()` позволяет автоматически заполнить гистограмму значениями из дерева. Для этого необходимо создать гистограмму и использовать оператор ″>>″ в первом аргументе метода:

```
TH1F *myh = new TH1F("myh","Energy deposition",100,0,300);
myt->Draw("energy>>myh");
```

Другие возможности при работе с деревьями ROOT

Выше перечислены только самые простые методы работы с деревьями ROOT. Этих сведений достаточно для выполнения практических заданий. Кратко перечислим, какие еще возможности доступны при работе с деревьями.

1. Класс `TChain` позволяет организовать одно виртуальное дерево из нескольких однотипных деревьев. Такая задача часто возникает при анализе больших объемов данных. Например, набор экспериментальной статистики может занимать долгое время – многие месяцы или даже годы. В таком случае процесс набора разбивается на интервалы разумной длительности. Пусть статистика для каждого интервала сохраняется в отдельном файле в виде дерева. Структура деревьев во всех файлах одинакова, но в них лежат разные независимые события. Тогда пользователь может создать объект `TChain`, передать ему список файлов и в дальнейшем работать с `TChain` так, как будто это единое дерево, в котором лежат события из всех файлов. Другими словами, `TChain` «склеивает» деревья «по вертикали».
2. Метод `TTree::AddFriend()` позволяет «склеить» деревья «по горизонтали», т.е. расширить список аргументов, доступных для каждого события. Например, пусть для каждого события в дереве лежат «сырые» данные эксперимента. Программа реконструкции обрабатывает эти данные, формирует данные более высокого уровня абстракции, например треки заряженных частиц, и сохраняет их в другом дереве. Как теперь для каждого события прочитать и сырые, и реконструированные данные, которые хранятся в разных файлах и в разных деревьях? Именно это и позволяет сделать метод `TTree::AddFriend()`.
3. Здесь был показан только способ работы с деревом в интерактивной среде ROOT. Однако есть множество удобных механизмов для работы с деревьями в пакетном режиме, что необходимо для автоматической обработки больших объемов данных. Наиболее удобный из этих механизмов – автоматическая генерация шаблона класса для обработки дерева с помощью методов `TTree::MakeClass()` или `TTree::MakeSelector()`. Шаблоны предоставляют всю инфраструктуру по доступу к данным, пользователь добавляет только свой алгоритм обработки.
4. В дереве можно хранить не только простые типы данных, но и целые структуры и даже объекты. В случае, если в дереве сохраняется массив, его длина может быть переменной.
5. Существует упрощенный вариант дерева, класс `TNtuple`, в котором можно хранить только простые вещественные параметры для каждого события. Другими словами, `TNtuple` представляет собой электронную таблицу, колонки в которой могут быть только вещественного типа. Для многих задач дерева такой простой структуры достаточно.
6. Деревья предоставляют удобные механизмы многомерного интерактивного анализа данных, например отображение их в параллельных координатах.

2 Программный пакет Geant4

2.1 Обзор

При проектировании эксперимента и во время последующей обработки набранных данных необходимо знать детальные характеристики различных параметров эксперимента, например эффективность регистрации, точность измерения различных величин, радиационную нагрузку и пр. Современный эксперимент в физике высоких энергий проводится с использованием детекторов, состоящих из набора самых разнообразных блоков вещества сложной формы, регистрирующих элементов и считывающей электроники. Для систем такого уровня сложности точность простых оценок совершенно недостаточна. Поэтому для корректного расчета нужных характеристик используется компьютерное моделирование – своего рода компьютерный эксперимент.

Рисунок 15. Основные стадии моделирования эксперимента в области физики высоких энергий

Способы организации компьютерного моделирования отличаются в различных областях знаний. На Рисунок 15 приведены стадии полного компьютерного моделирования, характерные для физики высоких энергий. Можно выделить следующие основные стадии.

1. Первой стадией работы является генерация первичных частиц, или первичное моделирование. Входными параметрами являются динамика изучаемых процессов, параметры эксперимента (например, положение пучка, разброс энергии и т. п.), в результате работы первичного генератора формируется набор частиц и их начальные кинематические параметры (координаты, время рождения, импульсы, поляризация).

2. После этого моделируется история существования этих частиц – распространение, взаимодействие с веществом, распады, поглощение веществом, то же самое делается для производных (вторичных) частиц. В результате, в активных элементах детектора оказываются зарегистрированными различные следствия этих взаимодействий – энерговыделение, число ионов, излученный черенковский свет и пр.

3. Используя информацию, полученную на предыдущей стадии, и построив модель работы электроники детектора, формируется отклик электроники детектора. Объединив отклик электроники со всех систем детектора, формируется событие, эквивалентное событию с реального детектора. После этой стадии, события моделирования имеют формат, эквивалентный событиям с детектора.

Описанные стадии повторяются многократно, в результате чего формируется множество статистически независимых событий. Полученные данные эквивалентны событиям, полученным с реальной установки. Как правило, к ним применимы те же процедуры калибровки и реконструкции, что и к реальным событиям. На финальной стадии производится анализ данных, эквивалентный анализу экспериментальных событий, при этом наличие дополнительной информации о заданных начальных частицах и истории их прохождения через детектор позволяет получить искомые параметры, необходимые для обработки реального эксперимента.

Программное обеспечения для моделирования эксперимента, как правило, очень сложное – оно должно включать математические модели взаимодействия всех возможных частиц со всеми элементами детектора. Часть программного обеспечения уникальна для каждого эксперимента – например, описание структуры эксперимента, модель работы электроники эксперимента и т.п. Однако значительная часть программы моделирования универсальна – в частности, модель взаимодействия частиц с веществом. Поэтому неудивительно, что существуют программные пакеты, в которых реализована универсальная часть программы моделирования и предоставлены возможности для добавления специфических для каждого эксперимента модулей.

В настоящее время для моделирования прохождения частиц через детектор повсеместно применяется программный пакет Geant4 [7]. В разработку этого пакета вложены усилия многих экспертов в области физики высоких энергий. В GEANT4 объединены существующие знания о взаимодействии различных частиц с разнообразными веществами, программно решены задачи распространения частиц в электромагнитных полях, задачи описания детектора, визуализации и многие другие. Пакет применяется не только в физике высоких энергий, но и в других областях, связанных с исследованием жестких излучений – медицина, контроль безопасности и др. Кроме собственно программного обеспечения, пакет содержит таблицы со свойствами различных веществ, доступные в открытой литературе, таблицы сечений взаимодействия стабильных частиц с веществом, другие данные.

Geant4 – это не программа, а набор программных инструментов (*toolkit*), с помощью которых пользователь может создать программу моделирования. На основе Geant4 разработано множество более специализированных программных каркасов.

Основная часть пакета, *ядро* Geant4, связывает воедино все программные модули (в текущей версии ядро состоит из 17 модулей). Основные задачи ядра: организация процесса моделирования, организация обмена данными между модулями, предоставление механизмов для работы с геометрическим представлением детектора, предоставление механизмов для управления моделями отдельных физических процессов. Пользователь должен описать *конструкцию детектора*, сгенерировать *первичные частицы* и передать управление ядру (Рисунок 16). Ядро моделирует взаимодействие этих частиц с детектором (включая моделирование вторичных частиц) и формирует *хиты*, описывающие единичные акты взаимодействия. Пользователь использует хиты при моделировании отклика электроники детектора.

Вся информация о Geant4 доступна на официальной веб-страничке [8]. Пакет Geant4 очень обширный и сложный. Для первоначального ознакомления с пакетом рекомендуется изучить руководство пользователя [9]. Более детальная информация об архитектуре пакета приведена в [10]. Описание основных доступных моделей взаимодействия частиц с веществом приведено в [11]. Вместе с исходным кодом пакета распространяется и обширный перечень примеров различного уровня сложности, которые незаменимы в качестве начального шаблона при разработке собственной программы моделирования.

Далее приводятся основные сведения о структуре Geant4 и способах работы с ним, которых достаточно для первоначального знакомства с пакетом и для выполнения практиче-

ских заданий. Предполагается, что читатель хорошо знаком с основами объектно-ориентированного программирования и с языком C++.

2.2 Алгоритм моделирования в Geant4

В Geant4 предполагается, что частица взаимодействует не с веществом, а с отдельными атомами вещества. По умолчанию предполагается, что центры взаимодействия (атомы, молекулы) распределены в пределах вещества равномерно и случайно, согласно известным макроскопическим параметрам вещества: химический и изотопный состав, плотность, фазовое состояние,…

Такое точечное описание неприменимо для моделирования когерентных процессов (например, распространение света в веществе, или распространение холодных и ультрахолодных нейтронов). В Geant4 возможно моделирование и таких процессов, но это требует дополнительных усилий.

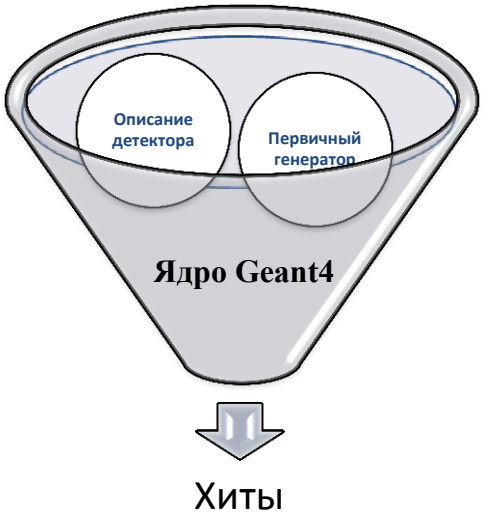


Рисунок 16. Ядро Geant4

Для расчетов используется метод Монте-Карло. Пусть существует только один механизм взаимодействия частицы с веществом. Вероятность того, что частица пройдет в веществе расстояние L до взаимодействия: $p(L) = \frac{1}{\lambda} \cdot \exp(-L/\lambda)$, где $\lambda = 1/n\sigma$ – длина свободного пробега, n – концентрация центров взаимодействия (атомов, молекул), σ – сечение взаимодействия. Концентрация зависит только от макроскопических свойств вещества. Сечение же зависит от микроскопической модели взаимодействия: от типа частицы, ее скорости, состава вещества и т.п.

Пусть взаимодействие является точечным, т.е. до взаимодействия частица проходит вещество без изменений, а в момент взаимодействия мгновенно изменяет свои параметры. Тогда прохождение частицы через вещество можно описать с помощью следующего итерационного алгоритма моделирования (Рисунок 17):

1) получаем начальные параметры частицы;
2) вычисляем n, σ и λ; длина свободного пробега зависит как от неизменных свойств вещества, так и от текущих параметров частицы (например, импульса);
3) генерируем случайное число $L^{(i)}$ согласно распределению $p(L)$;
4) проводим частицу на расстояние $L^{(i)}$;
5) моделируем акт взаимодействия;
6) если частица не исчезла, повторяем весь процесс.

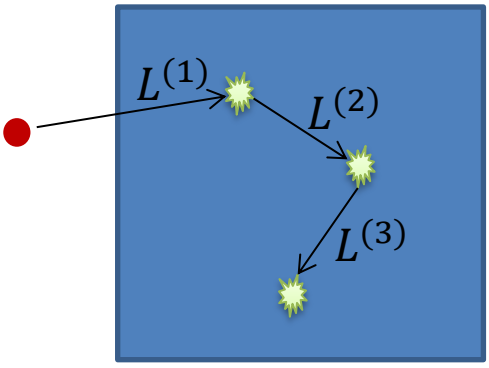


Рисунок 17. Моделирование точечного процесса методом Монте-Карло

Как правило, существует несколько конкурирующих механизмов взаимодействия частицы с веществом. Например, при прохождении -кванта может произойти комптоновское рассеяние, рождение e^+e^- пары или фотопоглощение. При наличии m независимых конку-

рирующих механизмов взаимодействия описанный алгоритм моделирования необходимо усложнить (Рисунок 18):

1) получаем начальные параметры частицы;
2) вычисляем длину свободного пробега λ_i $(i = 1 \dots m)$ для каждого типа взаимодействий;
3) генерируем случайные числа L_i согласно распределениям $p_i(L)$, т.е. для каждого типа взаимодействия генерируем случайную длину пробега в предположении, что других взаимодействий нет;
4) выбираем минимальное из предложенных чисел и проводим частицу на расстояние
$$L^{(i)} = \min L_j,\ j = 1 \dots m\ ;$$
5) моделируем акт того взаимодействия, который дал минимальное значение $L^{(i)}$ (не обязательно тот, у которого наименьшая длина свободного пробега λ_i);
6) если частица не исчезла, повторяем весь процесс.

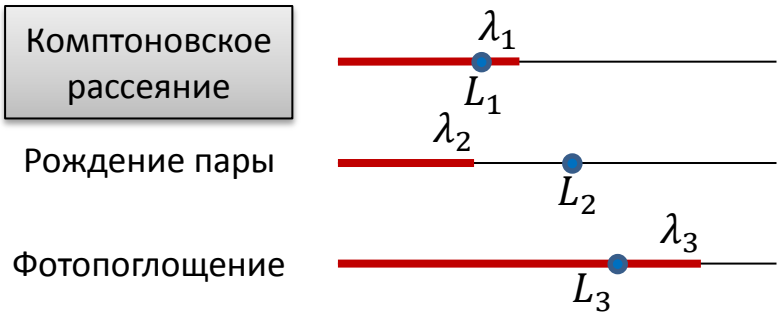


Рисунок 18. Алгоритм моделирования при наличии нескольких конкурирующих механизмов взаимодействия. Длина свободного пробега λ вычисляется на каждой итерации моделирования. Шаг моделирования L – случайное число с плотностью распределения $\sim exp(-L/\lambda)$.

Для того, чтобы описать алгоритм моделирования в Geant4, надо ввести еще одно усложнение. Кроме точечных типов взаимодействия существуют также и непрерывные взаимодействия, когда параметры частицы изменяются непрерывно при прохождении через вещество. К таким взаимодействиям относится, например, ионизация вещества, или поворот частицы при движении в магнитном поле. Конечно, непрерывное взаимодействие можно описать как серию очень слабых точечных взаимодействий с малой длиной пробега. Однако такое описание непрактично, т.к. требует слишком много вычислительных ресурсов. Вместо этого, в Geant4 используется следующая модификация описанного выше алгоритма:

1) для каждого точечного и непрерывного типа взаимодействия генерируется длина шага L_i;
2) выбирается минимальный из предложенных шагов и частица проводится на соответствующее расстояние; если выбранный шаг соответствует точечному взаимодействию, то моделируется акт этого взаимодействия;
3) моделируется эффект всех видов непрерывных взаимодействий на выбранной длине шага.

Для управления процессом моделирования, в Geant4 добавлены специальные типы точечных «взаимодействий», которые не меняют параметры частицы, но ограничивают длину шага моделирования (например, не позволяют шагу переходить границу между разными веществами). Кроме того, для каждой частицы определяются два типа точечных взаимодействий: для движущейся и для покоящейся частицы, т.к. соответствующие механизмы взаимодействия могут отличаться.

В рамках описанного алгоритма каждый тип взаимодействия частицы с веществом описывается своей собственной моделью. При описании одного механизма, нет необходимо-

сти учитывать, может ли частица участвовать в каких либо других взаимодействиях. Это позволяет организовать программный код удобным образом. Модель каждого типа взаимодействия представлена в виде отдельного класса, в котором реализованы два метода: `GetPhysicalInteractionLength()`, в котором генерируется длина очередного шага моделирования L_i, и `DoIt()`, в котором собственно моделируется акт взаимодействия. Ядро Geant4 использует эти два метода для реализации описанного алгоритма.

Библиотека Geant4 содержит большое количество классов – моделей различных типов взаимодействия частиц с веществом.

2.3 Организация программы моделирования в Geant4

Организация процесса моделирования в Geant4 описывается иерархией ***заход-событие-трек-шаг.***

Заход – это множество событий, моделирование которых производилось в аналогичных условиях. При этом отдельные события в одном заходе статистически независимы. В пределах одного захода пользователь не может изменять геометрию детектора или изменять список или параметры возможных типов взаимодействий.

Понятие захода в Geant4 полностью аналогично понятию захода в эксперименте, означающего множество событий, зарегистрированных при одинаковых условиях набора данных. В начале захода ядро Geant4 выполняет различные подготовительные действия:

1) оптимизирует геометрическое представление детектора в памяти программы;
2) вычисляет разнообразные сечения для возможных типов взаимодействий.

Событие – это коллекция всей информации, полученной при моделировании всех частиц, соответствующих одному запуску первичного генератора.

Процесс моделирования одного события организован следующим образом. Моделирование каждого события начинается с запуска первичного генератора, который создает первичные частицы и помещает их в стек. Ядро Geant4 берет из стека по одной частице и моделирует ее прохождение через вещество детектора. Если в процессе моделирования появились вторичные частицы, они помещаются в стек. Моделирование события заканчивается, когда в стеке не остается ни одной частицы.

По окончании моделирования одного события формируются:

1) коллекция первичных частиц и вершин;
2) коллекция хитов;
3) коллекция срабатываний чувствительных элементов детектора;
4) коллекция траекторий (промежуточных положений частиц).

Трек – это мгновенное состояние одной частицы. В любой конкретный момент трек обладает рядом характеристик: положение, импульс,… По мере движения частицы характеристики трека меняются. Трек создается или в первичном генераторе, или при рождении новой частицы в результате взаимодействия существующих частиц с веществом. Трек удаляется в следующих случаях:

1) частица вылетает из области моделирования (world volume);
2) частица «умирает» (распадается, поглощается,…);
3) частица останавливается и для нее не определено взаимодействий для покоящейся частицы.

По окончанию моделирования события ни одного трека не остается.

Минимальной единицей моделирования в Geant4 является ***шаг*** (step) моделирования. Шаг описывается двумя точками: началом и концом. Шаг содержит всю информацию о том, что произошло с частицей на протяжении шага: потеря энергии, время пролета, какой тип взаимодействия произошел,… Шаг не может пересекать границу геометрического элемента; в этом случае шаг заканчивается ровно на границе и конечная точка будет соответствовать уже новому геометрическому элементу.

Для взаимодействия пользователя и ядра Geant4 используется механизм объектно-ориентированного программирования – *наследование*. Все элементы моделирования, в которые может потребоваться вмешательство пользователя, выделены в виде *базовых* классов (Рисунок 19). При необходимости, пользователь реализует *производный* класс, в который добавляет необходимую функциональность. Ядро Geant4 работает с пользовательскими классами через интерфейс базового класса. Те элементы, которые обязательно должны быть определены пользователем (например, описание геометрии детектора), выделены в ядре Geant4 в виде *абстрактных* базовых классов.

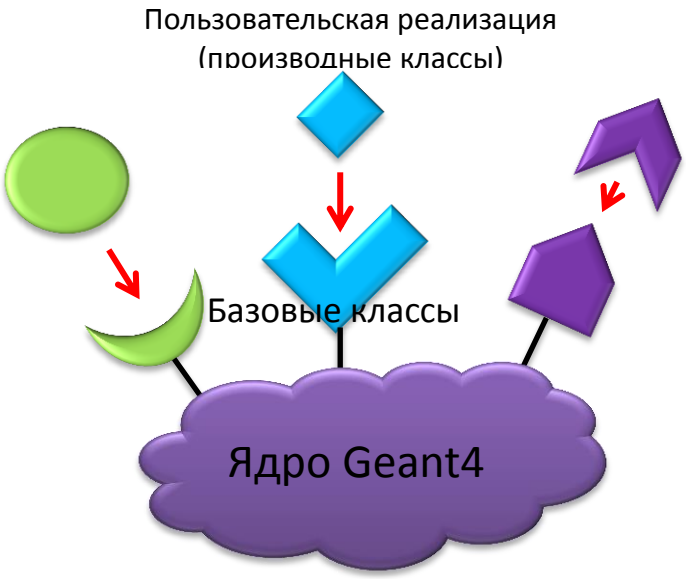


Рисунок 19. Взаимодействие ядра Geant4 и пользователя

Пользователь должен реализовать (создать класс-наследник, в котором реализованы абстрактные методы базового класса) следующие абстрактные базовые классы:

- `G4VUserDetectorConstruction` – описание геометрии детектора.
- `G4VUserPhysicsList` – создание списка возможных процессов взаимодействия.
- `G4VUserPrimaryGeneratorAction` – первичный генератор.

Пользователь может управлять процессом моделирования, выполняя необходимые действия на различных стадиях моделирования. Для этого необходимо реализовать следующие базовые классы:

- `G4UserRunAction` – действия перед началом захода и по окончании захода.
- `G4UserEventAction` – действия перед началом и по концу моделирования одного события.
- `G4UserTrackingAction` – действия перед началом и по концу моделирования одной частицы (трека).
- `G4UserStackingAction` – управление порядком моделирования вторичных частиц.
- `G4UserSteppingAction` – действия на каждом шаге моделирования.

Поскольку Geant4 – это не программа, а набор программных инструментов (библиотека), пользователь должен написать основную программу `main()`. Ниже приведен типичный пример `main()` в программе моделирования, основанной на Geant4:

```
main() {
  …
  // Construct the default run manager
  G4RunManager* runManager = new G4RunManager;
```

```
  // Set mandatory user initialization classes
  MyDetectorConstruction* detector = new MyDetectorConstruction;
  runManager->SetUserInitialization(detector);
  runManager->SetUserInitialization(new MyPhysicsList);
  // Set mandatory user action classes
  runManager->SetUserAction(new MyPrimaryGeneratorAction);
  // Set optional user action classes
  MyEventAction* eventAction = new MyEventAction();
  runManager->SetUserAction(eventAction);
  MyRunAction* runAction = new MyRunAction();
  runManager->SetUserAction(runAction);
  // Initialize G4 kernel and run one event
  runManager->Initialize();
  runManager->BeamOn(1);
  delete runManager();
}
```

Пояснения к примеру.

1) Создается экземпляр ядра Geant4 `runManager`, который управляет всем процессом моделирования.
2) Создаются экземпляры обязательных классов, которые должен реализовать пользователь: `MyDetectorConstruction`, `MyPhysicsList` и `MyPrimaryGeneratorAction`. Эти объекты передаются ядру с помощью вызова `SetUserInitialization()` или `SetUserAction()`. Так ядро Geant4 узнает о том, какие пользовательские функции оно должно вызвать для описания геометрии эксперимента, для описания модели взаимодействий и для генерации первичных частиц.
3) Создаются экземпляры классов, которые пользователь может реализовать для управления процессом моделирования на отдельных стадиях: здесь `MyEventAction` и `MyRunAction`. Эти объекты передаются ядру с помощью вызова `SetUserAction()`. Теперь ядро Geant4 перед началом и по концу моделирования захода и каждого события будет вызывать пользовательские функции, в которых, например, открывается файл, сбрасываются счетчики и т.п.
4) Ядро инициализируется с помощью `Initialize()`.
5) Проводится моделирование одного события с помощью `BeamOn(1)`.

Помимо описанных действий, в `main()` пользователь может инициализировать интерактивный текстовый или графический интерфейс.

2.4 Описание геометрии детектора

Для описания геометрии детектора, пользователь должен создать класс, производный (наследующий) от абстрактного базового класса `G4VUserDetectorConstruction`, в котором необходимо реализовать метод `Construct()`:

File MyDetectorConstruction.hh:

```
class MyDetectorConstruction : public G4VUserDetectorConstruction {
   public:
   …
   G4VPhysicalVolume* Construct();
   …
}
```

File MyDetectorConstruction.cc:

```
G4VPhysicalVolume* MyDetectorConstruction::Construct()
{
  // Implementation of the experimental set-up
  …
}
```

В этом методе описывается вся геометрия эксперимента:

1) определяются все используемые вещества;
2) определяются все геометрические объемы, из которых состоит детектор;
3) определяются и размещаются все элементы детектора;
4) определяются чувствительные элементы детектора;
5) формируется карта электромагнитных полей;
6) определяются параметры визуализации элементов детектора.

Существует множество способов определения вещества, из которого состоят элементы детектора: как изотоп, химический элемент, молекула, или любая смесь перечисленных вариантов. В Geant4 предопределено большое количество широко используемых веществ. Ниже приведены примеры определения двух веществ (газообразного Xe и пластика C_9H_{10}):

```
PVPhysicalVolume*  MyDetectorConstruction::Construct()
{
  …
  // Define Xenon Gas
  density = 5.458*mg/cm3;
  pressure = 1*atmosphere;
  temperature = 293.15*kelvin;
  G4Material* xenon = new G4Material(name="XenonGas", z=54.,
                                     a=131.29*g/mole, density,
                                     kStateGas, temperature, pressure);

  // Define C9H10 scintillator
  G4Element* H  = new G4Element(name="Hydrogen", symbol="H", z=1.,
                                     a=1.01*g/mole);
  G4Element* C  = new G4Element(name="Carbon", symbol="C", z=6.,
                                     a=12.01*g/mole);
  G4Material* scintillator = new G4Material(name = "Scintillator",
                                            density = 1.032*g/cm3,
                                            numberOfComponents=2);
  scintillator->AddElement(C, numberOfAtoms = 9);
  scintillator->AddElement(H, numberOfAtoms = 10);
  ...
}
```

Описание собственно геометрии элемента детектора производится на трех уровнях:

1) описание геометрической формы и размера элемента – класс `G4VSolid`;
2) описание логического объема, для которого добавляется информация о веществе, из которого состоит элемент, наличии магнитного и электрического полей, пассивности или активности элемента,… – класс `G4LogicalVolume`;
3) описание физического объема, представляющего собой логический объем,

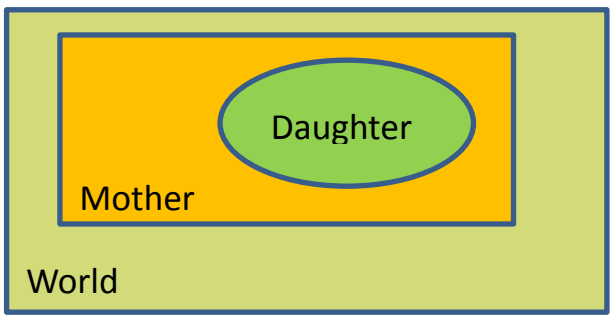


Рисунок 20. Иерархия геометрических элементов в Geant4

помещенный в конкретную область пространства детектора – класс `G4VPhysicalVolume`.

Элементы детектора не могут пересекаться частично, но один объем (материнский) может полностью содержать другой (дочерний). Существует выделенный элемент детектора, «мир» (*world volume)*, который содержит все остальные элементы (Рисунок 20). Особенность

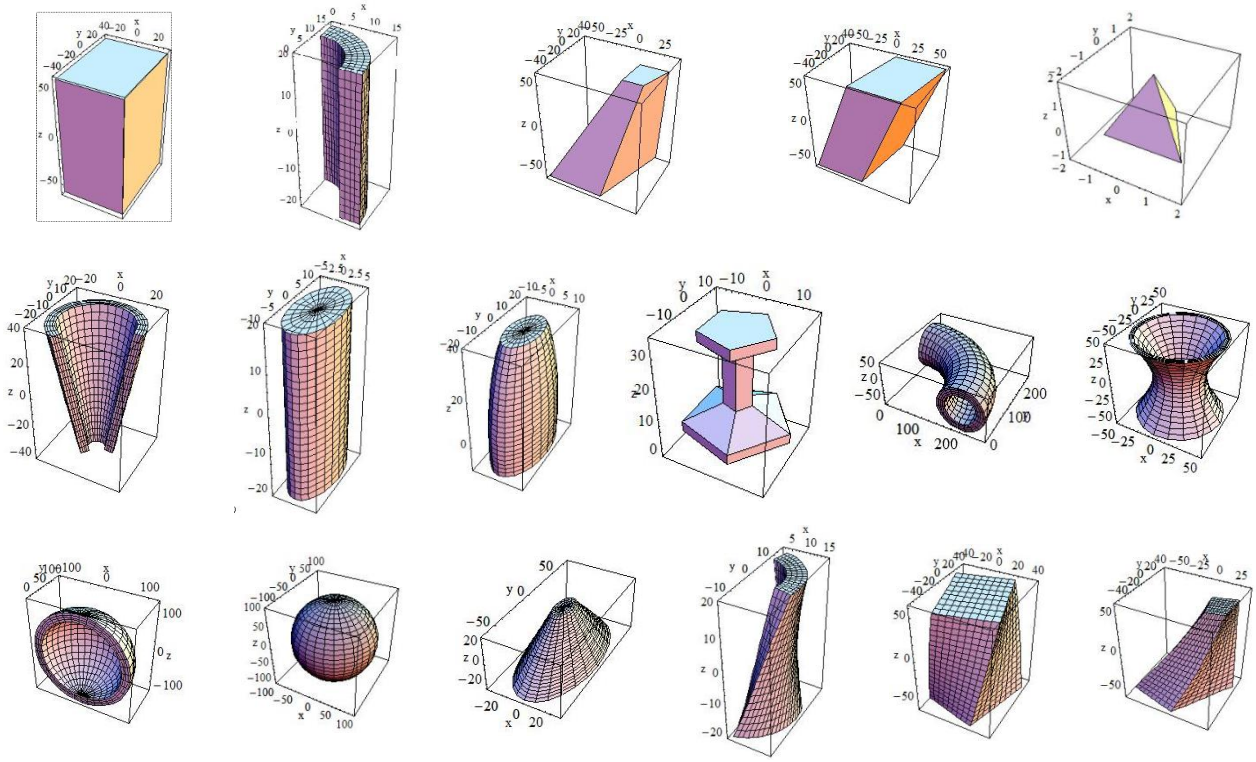


Рисунок 21. Примеры элементарных геометрических объемов, доступных в Geant4

этого элемента состоит в том, что у него нет материнского объема.

В Geant4 предопределено значительное число элементарных объемов, с помощью которых можно описать форму элементов детектора (Рисунок 21). Кроме того, можно использовать форму, получающуюся в результате объединения или пересечения элементарных объемов. В случае сложных форм, их можно описать с помощью пространственной сетки.

Ниже приведен пример описания простого детектора, состоящего из кубического «мира» и помещенной в него кубической мишени.

```
PVPhysicalVolume*  MyDetectorConstruction::Construct()
{
  …
 // Создаем "мир": сначала форма (куб)
 solidWorld = new G4Box("World", worldHL, worldHL, worldHL);
 // … теперь из чего состоит
 logicWorld = new G4LogicalVolume(solidWorld,air,"World",0,0,0);
 // … помещаем его на арену
 physicalWorld = new G4PVPlacement(
                      0,               // no rotation
                      G4ThreeVector(), // at (0,0,0)
                      logicWorld,      // its logical volume
                      "World",         // its name
                      0,               // its mother  volume
                      false,
```

```
                    0);

 // Создаем мишень и помещаем ее в «мир»
 solidTarget = new G4Box("Target", targetHL, targetHL, targetHL);
 logicTarget = new G4LogicalVolume(solidTarget, targetMaterial,
                   "Target",0,0,0);
 physicalTarget = new G4PVPlacement(
                    0,               // no rotation
                    positionTarget,  // at (x,y,z)
                    logicTarget,     // its logical volume
                    "Target",        // its name
                    logicWorld,      // its mother  volume
                    false,
                    0);
  ...
}
```

2.5 Описание модели взаимодействия частиц с веществом

По умолчанию в Geant4 не «включены» никакие типы частиц и взаимодействий. Пользователь формирует списки частиц и для каждой частицы перечисляет типы возможных взаимодействий. Для этого пользователь должен создать класс, производный (наследующий) от абстрактного базового класса `G4VUserPhysicsList`, в котором необходимо реализовать методы `ConstructParticles()`, `ConstructProcesses()` и `SetCuts()`:

File MyPhysicsList.hh:

```
class MyPhysicsList : public G4VUserPhysicsList {
   public:
   …
   void ConstructParticles();
   void ConstructProcesses();
   void SetCuts();
   …
}
```

File MyPhysicsList.cc:

```
// Define all particles, which can occur in simulation
void  MyPhysicsList::ConstructParticles()
{
 G4Electron::ElectronDefinition();
 G4Positron::PositronDefinition();
 G4Gamma::GammaDefinition();
}

// List, what can happen to these particles
void  MyPhysicsList::ConstructProcesses(){
 …
 if (particleName == "gamma") {
    pManager->AddDiscreteProcess(new G4PhotoElectricEffect());
    pManager->AddDiscreteProcess(new G4ComptonScattering());
    pManager->AddDiscreteProcess(new G4GammaConversion());
 } else if (particleName == "e-") {
    pManager->AddProcess(new G4MultipleScattering(), -1, 1,1);
    pManager->AddProcess(new G4eIonisation(), -1, 2,2);
```

```
    pManager->AddProcess(new G4eBremsstrahlung(), -1,-1,3);
 }
  …
}
```

Как правило, нет необходимости подробно перечислять все возможные частицы и процессы. Geant4 предоставляет множество уже готовых наборов частиц/процессов, которые можно использовать при реализации этих функций.

2.6 Первичный генератор

Последний обязательный элемент программы моделирования в рамках Geant4 – это первичный генератор, который задает начальный список частиц и их параметры для каждого события. С точки зрения Geant4, первичный генератор – это внешний элемент, который зависит от того, что именно моделируется. Например, для моделирования фоновых условий в детекторе может понадобиться первичный генератор космических событий, а для моделирования сигнала – генератор событий, рожденных в e^+e^- столкновениях.

Для реализации первичного генератора пользователь должен создать класс, производный (наследующий) от абстрактного базового класса `G4VUserPrimaryGeneratorAction`, в котором необходимо реализовать методы `GeneratePrimaries()`, в котором создать (положить в стек) все начальные частицы для данного события. Для каждой частицы задается ее тип, координаты и импульс (при необходимости – и другие параметры, например, спин). Пример простого первичного генератора, в котором в каждом событии есть только одна начальная частица, электрон, с одними и теми же начальными параметрами:

File MyPrimaryGenerator.hh:

```
class MyPrimaryGenerator : public G4VUserPrimaryGeneratorAction {
   public:
   …
   void GeneratePrimaries(G4Event*);
…
}
```

File MyPrimaryGenerator.cc:

```
MyPrimaryGenerator::MyPrimaryGenerator()
{

  G4int numberOfParticles = 1;
  particleGun = new G4ParticleGun (numberOfParticles);

  G4ParticleTable* partTable = G4ParticleTable::GetParticleTable();
  G4ParticleDefinition* particle = partTable->FindParticle("e-");

  particleGun->SetParticleDefinition(particle);
  particleGun->SetParticlePosition(G4ThreeVector(x,y,z));
  particleGun->SetParticleMomentumDirection(G4ThreeVector(x,y,z));
  particleGun->SetParticleEnergy(energy);

}

void MyPrimaryGenerator::GeneratePrimaries(G4Event* anEvent)
{
```

```
    particleGun->GeneratePrimaryVertex(anEvent);
}
```

2.7 Взаимодействие с ядром Geant4 в процессе моделирования

Для взаимодействия с ядром Geant4 в процессе моделирования пользователь может реализовать классы, производные (наследующие) от перечисленных ниже базовых классов. Тогда в указанные контрольные моменты (начало и конец моделирования захода, события, и т.п.) ядро Geant4 вызовет пользовательскую, а не встроенную, реализацию соответствующего метода.

- Базовый класс `G4UserRunAction`: перед началом моделирования захода вызывается метод `BeginOfRunAction(const G4Run*)`, по концу моделирования захода – метод `EndOfRunAction(const G4Run*)`. Типичное использование этих методов: создание гистограмм и деревьев ROOT, открытие файла, сохранение результатов.
- Базовый класс `G4UserEventAction`: перед началом моделирования каждого события вызывается метод `BeginOfEventAction(const G4Event*)`, по концу моделирования каждого события – метод `EndOfEventAction(const G4Event*)`. Типичное использование этих методов: обнуление счетчиков, анализ данных для одного события, объединение результатов прохождения различных частиц через один и тот же элемент детектора.
- Базовый класс `G4UserTrackingAction`: перед началом моделирования каждой частицы (первичной или вторичной) вызывается метод `PreUserTrackingAction(const G4Track*)`, по концу моделирования частицы – метод `PostUserTrackingAction(const G4Track*)`. Типичное использование этих методов: удаление неинтересных треков без их моделирования, формирование графа первичных и вторичных частиц.
- Базовый класс `G4UserSteppingAction`: на каждом шаге после моделирования очередного акта взаимодействия вызывается метод `UserSteppingAction(const G4Step*)`. Типичное использование этого метода: сохранение промежуточных данных.

2.8 Сохранение результатов моделирования

В процессе моделирования Geant4 производит большое количество информации: списки частиц, множество шагов моделирования с подробным описанием, что произошло на каждом шаге, и т.п. Что делать с информацией, полученной во время моделирования, как ее отфильтровать и сохранить? Geant4 предоставляет несколько механизмов сохранения результатов.

1. Используя `UserSteppingAction()`, самостоятельно сохранять всю интересующую информацию. Например, проверять на каждом шаге, не произошло ли выделение энергии в чувствительной области детектора. Если произошло, просуммировать энерговыделение по всем шагам для каждого независимого элемента детектора. Это самый простой подход, но все требуется писать самостоятельно.
2. Использовать встроенный в Geant4 механизм сохранения результатов на сетке (*scoring mesh*). Удобный механизм для расчета поглощенной дозы, потоков частиц.
3. Использовать встроенный механизм работы с активными областями детектора `G4VSensitiveDetector`. Наиболее гибкий механизм. Используется для детального моделирования, включая моделирование работы электроники.

Как правило, окончательные результаты моделирования события сохраняются в том же формате, что и реальные данные эксперимента, дополненные информацией об исходных параметрах моделирования (например, списком первичных частиц).

3 Практические задания

3.1 Toy Monte-Carlo

Постановка задачи

Часто требуется оценить статистическую и систематическую ошибки выбранной процедуры анализа данных. В простых случаях, например, когда все распределения являются нормальными и используются хорошо известные статистические приемы, существуют аналитические формулы для подобных оценок. Однако в более сложных случаях необходимо использовать моделирование методом Монте-Карло.

Одна из наиболее распространенных процедур такого моделирования – toy Monte-Carlo, «простое» моделирование. В рамках этой процедуры исходные распределения для сырых данных считаются известными. По заданным распределениям генерируется массив сырых данных, которые затем проходят выбранную процедуру анализа и ее результаты сравниваются с параметрами, заложенными в моделирование. Как правило, генерируется множество независимых массивов данных, и из распределения полученных результатов извлекается информация о статистической ошибке и смещении процедуры.

Более сложными процедурами являются полное или параметризованное моделирование методом Монте-Карло, в котором сырые данных генерируются с использованием математических моделей, учитывающих лежащие в основе физические процессы. Пакет Geant4 предназначен для моделирования именно такого типа. Как правило, такие процедуры требуют гораздо больше вычислительных ресурсов, и они используются для оценки влияния параметров детектора на результаты анализа. Для оценки статистических свойств анализа достаточно toy Monte-Carlo.

В задании требуется провести небольшое исследование с помощью простого моделирования методом Монте-Карло.

1. Необходимо написать генератор случайных чисел, распределенных согласно плотности, заданной преподавателем. Следует использовать метод обратной трансформации или комбинированный метод
2. Продемонстрировать свойства двух различных оценок положения распределения: выборочного среднего и выборочной медианы. Пусть в одном эксперименте генерируется n_{data} (типичное значение 100) величин, распределенных согласно заданному распределению. По полученной выборке определяются значения выборочного среднего и выборочной медианы. Для того, чтобы изучить свойства этих двух оценок, повторим процедуру для n_{exp} (типичное значение 1000) экспериментов. Из распределения полученных значений оценок для различных экспериментов необходимо извлечь следующую информацию.
    1) Оценить мат.ожидание заданного распределения, определить точность этой оценки и сравнить полученное значение с расчетным.
    2) Оценить асимптотическую эффективность выборочной медианы в сравнении с выборочным средним. Для этого нужно провести моделирование для разных значений n_{data} и определить, чему асимптотически равно отношение дисперсий двух оценок.

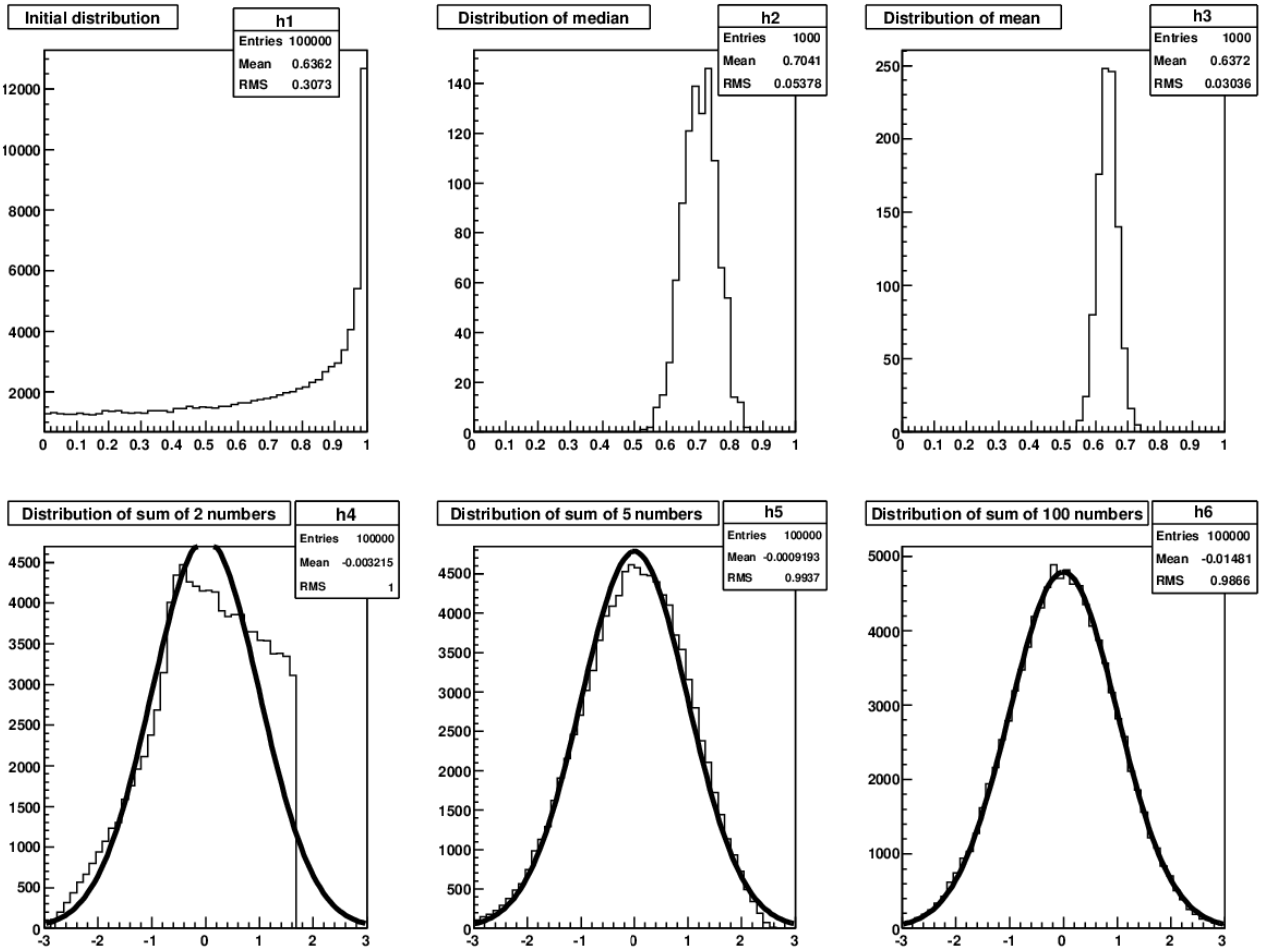


Рисунок 22. Пример отчета по первому заданию

3) Продемонстрировать робастность выборочной медианы. Для этого модифицируйте генератор случайных чисел, добавив в него небольшую вероятность «выбросов». Покажите, как смещение и дисперсия двух оценок зависят от доли выбросов.

3. Продемонстрируйте предсказание центральной предельной теоремы. Для этого сгенерируйте распределение нормированной суммы 2, 5 и 100 случайных величин, подчиняющихся заданному распределению. Сходится ли полученное распределение к нормальному?

Указания

К заданию прилагается пример, в котором генерируется равномерное распределение, вычисляется выборочное среднее и демонстрируется центральная предельная теорема для суммы 2 случайных чисел. Этот пример может быть использован в качестве каркаса для написания программы. Необходимо модифицировать данный пример следующим образом.

1) Переписать функцию `getRandom()` так, чтобы она генерировала случайную величину, распределенную согласно заданной плотности.
2) Заменить значения `x0` и `sig0` на мат.ожидание и дисперсию заданного распределения.
3) Реализовать вычисление выборочной медианы.

4) Набрать гистограммы со значениями нормированных сумм 5 и 100 случайных величин, аналогично тому, как это сделано в случае суммы 2 величин. Сравнить полученные распределения с нормальным.

В качестве отчета необходимо подготовить картинку, показанную на Рисунок 22.

Дополнительные вопросы

1. При каком уровне выбросов стоит использовать выборочную медиану вместо выборочного среднего?
2. Сколько нужно сложить случайных величин, чтобы 95% доверительный интервал, определенный в предположении нормального распределения, был правильным с заданной точностью?
3. Замените свое распределение распределением Коши. Работает ли в этом случае центральная предельная теорема?
4. Вычислите доверительный интервал для параметров `Mean` (выборочное среднее) и `RMS` (выборочная дисперсия) для распределения h3 на Рисунке 22, зная свойства исходного распределения и постановку эксперимента. Попали ли значения этих параметров, полученные в вашем численном эксперименте, в полученные интервалы?

3.2 Моделирование детектора

Постановка задачи

Требуется смоделировать отклик детектора элементарных частиц. Детектор представляет собой простой многослойный калориметр, показанный на Рисунок 23. В целом, калориметр представляет собой либо куб, либо цилиндр, разделенный на N слоев ($N = 10$).

В качестве активного вещества используются широко распространенные в калориметрии материалы: кристаллы NaI, CsI, BGO, LSO, сжиженные благородные газы Ar, Kr, Xe. Каждый студент использует индивидуальный активный материал, предложенный преподавателем.

Регистрируемые частицы влетают в калориметр вдоль оси детектора. Возможны 3 типа частиц: электроны e, мюоны μ и гамма-кванты γ. Начальная энергия всех частиц одинакова, тип частицы должен выбираться случайным образом для каждого события.

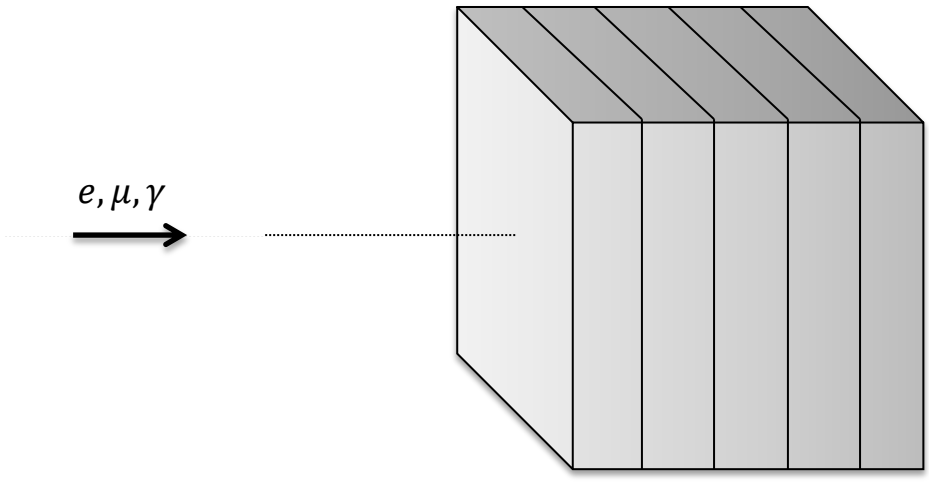


Рисунок 23. Геометрия детектора

Программа моделирования должна быть написана таким образом, что следующие параметры могут быть легко изменены без значительного переписывания кода: размеры детектора, количество слоев, активное вещество, энергия влетающих частиц.

В качестве результата необходимо провести моделирование 10000 событий. Для каждого события необходимо сохранить: тип частицы, полное энерговыделение в калориметре, энерговыделение в каждом слое. Результаты моделирования в форме дерева ROOT необходимо сохранить, т.к. эти данные будут использованы в последующих заданиях.

Указания

К заданию прилагается пример, в котором описанная задача решена для однослойного калориметра и одного типа частиц. Программный код состоит из основной программы (файл `exampleN03.cc`) и файлов с описанием (директория inc/) и реализацией (директория src/) пользовательских классов. В примере реализованы следующие классы.

1. Класс `ExN03DetectorConstruction`: описание геометрии детектора. Этот класс необходимо модифицировать следующим образом:
    1) усложнить описание геометрии детектора, разбив единый активный объем калориметра на N слоев; поскольку во время моделирования необходимо будет идентифицировать каждый слой, им удобно присвоить уникальный целочисленный идентификатор; это можно сделать, используя аргумент `copyNo` конструктора `G4PVPlacement`;
    2) изменить активное вещество калориметра на вещество, заданное преподавателем.
2. Класс `ExN03PhysicsList`: задание списка частиц и описание модели взаимодействия частиц с веществом. Не требует модификации.
3. Класс `ExN03PrimaryGeneratorAction`: первичный генератор. Этот класс необходимо модифицировать, добавив случайный выбор одного из 3 типов частиц. Обратите внимание, что по условиям задачи полная энергия для всех типов частиц одинакова. Однако при создании частицы в первичном генераторе задается ее *кинетическая* энергия, которая отличается для трех типов частиц из-за их разной массы покоя.
4. Класс `ExN03RunAction`: действия по началу и по концу захода. В данном случае моделируется один заход, состоящий из 10000 событий. По началу захода открывается файл в формате ROOT и в нем создается дерево. По концу захода дерево записывается в файл, и файл закрывается. Этот класс необходимо модифицировать, изменив структуру выходного дерева: необходимо добавить ветку для хранения типа частиц и ветку (массив) для хранения энерговыделений в каждом слое.
5. Класс `ExN03EventAction`: действия по началу и по концу моделирования одного события. Во время моделирования одного события в каждом слое калориметра может оказаться множество шагов моделирования. В поставленной задаче физически осмысленная величина – это суммарное энерговыделение из всех этих шагов. В классе `ExN03EventAction` создается счетчик выделенной энергии. Перед началом моделирования события он сбрасывается, а по концу суммарное энерговыделение записывается в дерево ROOT. Этот класс необходимо модифицировать, изменив алгоритм подсчета энерговыделения: необходимо вычислять суммарное энерговыделение для каждого слоя отдельно.
6. Класс `ExN03SteppingAction`: доступ к информации на каждом шаге моделирования. На каждом шаге проверяется, не выделилась ли энергия в чувствительной области детектора (калориметре), и если выделилась, то ее величина передается в

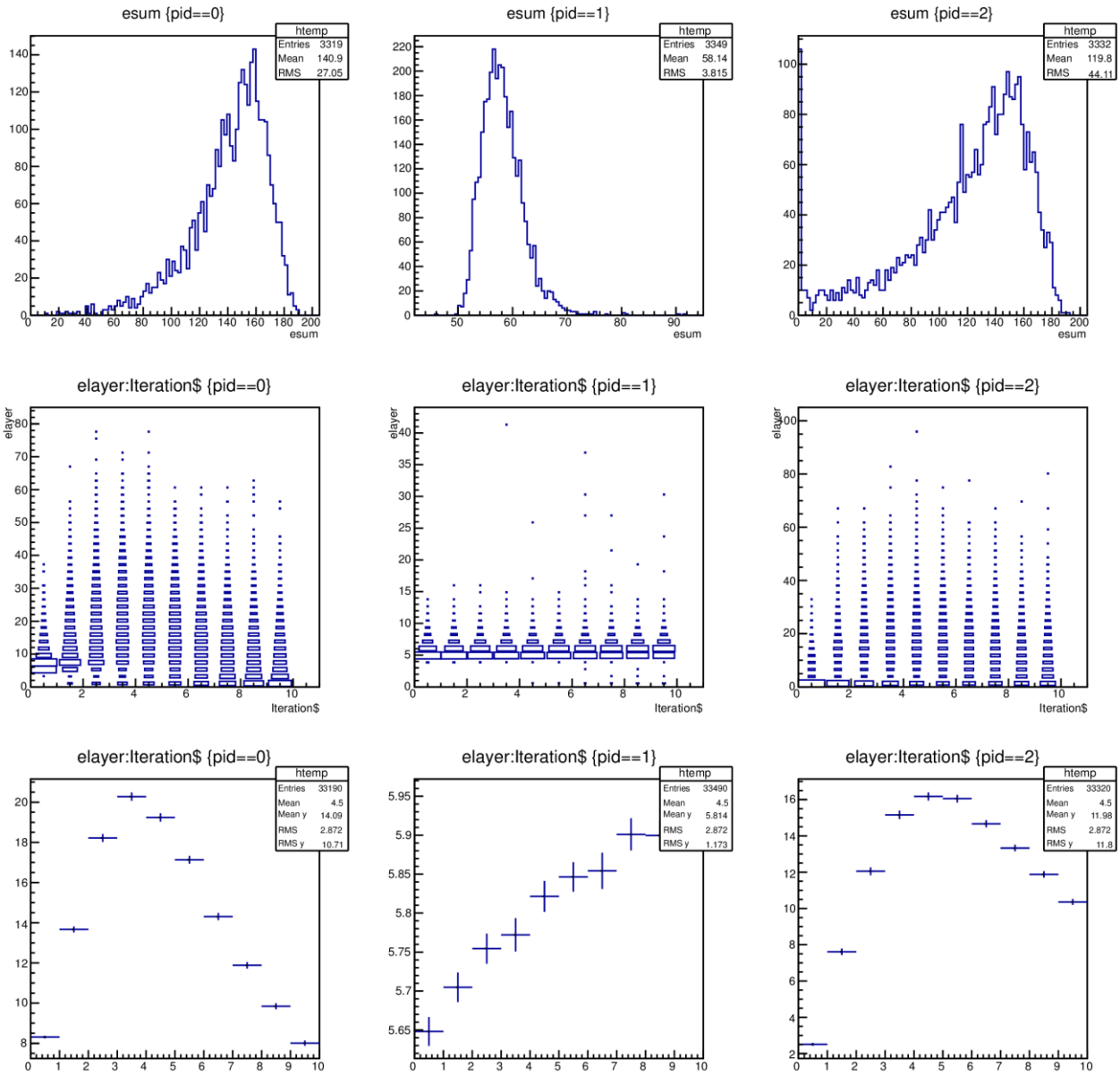


Рисунок 24. Пример результата моделирования

`ExN03EventAction` для накопления. Этот класс необходимо модифицировать, добавив проверку того, в каком слое произошло энерговыделение.

Перед началом работы с примером, необходимо установить переменные окружения:

```
declare -x G4WORKDIR="/home/{username}/work"
source /home/logashenko/geant4/geant4.9.5-install/share/
       Geant4-9.5.0/geant4make/geant4make.sh
source /home/logashenko/root/root/bin/thisroot.sh
declare -x G4ABLADATA="/home/logashenko/geant4/data/G4ABLA3.0"
declare -x G4LEDATA="/home/logashenko/geant4/data/G4EMLOW6.23"
declare -x G4LEVELGAMMADATA=
          "/home/logashenko/geant4/data/PhotonEvaporation2.2"
declare -x G4NEUTRONHPDATA="/home/logashenko/geant4/data/G4NDL0.2"
declare -x G4RADIOACTIVEDATA=
```

```
            "/home/logashenko/geant4/data/RadioactiveDecay3.4"
declare -x G4REALSURFACEDATA=
            "/home/logashenko/geant4/data/RealSurface1.0"
```

Рекомендуется записать эти команды в файл `env.sh`, и выполнять его перед началом работы:

```
source env.sh
```

Собрать пример можно с помощью команды gmake. Запуск программы моделирования:

```
$G4WORKDIR/ bin/Linux-g++/exampleN03 {command_file}
```

Если указано имя командного файла (в примере прилагается файл `run1.mac`), то программа будет запущена в пакетном режиме: будут выполнены все команды из файла и программа автоматически остановится. Этот режим следует использовать для моделирования большого числа событий. Если имя файла отсутствует, программа будет запущена в интерактивном режиме и каждое событие будет отображаться на экране. Для запуска моделирования N событий используйте команду

```
/run/beamOn N
```

Результатом моделирования является файл с данными в формате ROOT. На основе этого файла необходимо подготовить картинку, в которой для каждого типа частиц показаны:

1) полное энерговыделение в калориметре
   `t->Draw("esum","pid==1")`,
2) распределение энерговыделения в каждом слое
   `t->Draw("elayer:Iteration$","pid==1","box")`
3) и среднее энерговыделение в каждом слое
   `t->Draw("elayer:Iteration$","pid==1","prof")`.

В списке приведены команды ROOT, с помощью которых можно сформировать соответствующую картинку. Пример результата моделирования показан на Рисунок 24.

Основные и дополнительные вопросы

1. Объясните на качественном уровне отличия между полученными распределениями для разных типов частиц.
2. Попробуйте изменить список возможных взаимодействий для какого либо типа частиц в `ExN03PhysicsList` и посмотрите, как изменяться полученные распределения. Объясните результаты на качественном уровне.

3.3 Метод максимального правдоподобия

Постановка задачи

Пусть детектор регистрирует частицы двух типов и для каждой частицы измеряется только полное энерговыделение. Необходимо с помощью метода максимального правдоподобия оценить число событий каждого типа, анализируя измеренное распределение полного энерговыделения.

Из массива экспериментальных данных, сгенерированного в рамках задания 3.2, необходимо сформировать гистограмму полного энерговыделения в калориметре для событий двух типов (например, e и μ). Данная гистограмма (Рисунок 25) представляет собой модель экспериментальных данных.

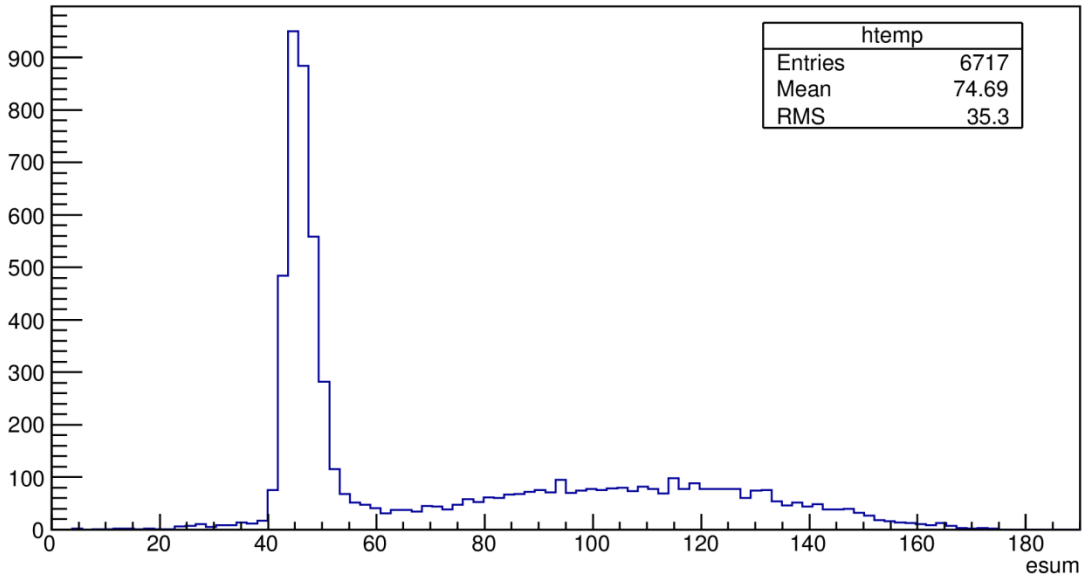


Рисунок 25. Пример экспериментальных данных: энерговыделение электроном и мюонов.

В экспериментальных данных перемешаны энерговыделения двух типов событий. Для того, чтобы оценить число событий каждого типа, необходимо подогнать данную гистограмму функцией

$$f(x) = n_e f_e(x) + n_\mu f_\mu(x)$$

где x – энерговыделение, n_k – число событий -того типа, $f_k(x)$ – плотность распределения энерговыделения для событий k-того типа. Подгонку необходимо реализовать методом максимального правдоподобия. По результатам подгонки необходимо получить оценку числа событий каждого типа и точность этой оценки.

Указания

Для того, чтобы использовать формулу для $f(x)$, необходимо выбрать аналитическую форму для функций $f_e(x)$ и $f_\mu(x)$. Для этого надо сформировать отдельную гистограмму для каждого типа событий (для моделированных данных это легко сделать) и подобрать соответствующую аналитическую функцию.

В данной задаче известно, что форма функций может быть описана как суперпозиция нескольких элементарных функций:

$$f(x) = p \cdot g_1(x) + (1-p) \cdot g_2(x)$$

где в качестве функций $g_i(x)$ можно использовать нормальное распределение

$$g(x; x_0, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-x_0)^2}{2\sigma^2}\right)$$

или вариант логарифмически-нормального распределения

$$g_l(x; x_0, \sigma, \eta) = \frac{1}{\sqrt{2\pi}\sigma} \cdot \frac{\eta}{\sigma_0} \cdot exp\left\{-\frac{1}{2}\left[\frac{ln^2\left(1 - \frac{x-x_0}{\sigma}\eta\right)}{\sigma_0^2} + \sigma_0^2\right]\right\},$$

где $\sigma_0 = \ln\left(\eta\sqrt{2\ln 2} + \sqrt{1+\eta^2 \cdot 2\ln 2}\right)/\sqrt{2\ln 2}$

Важно помнить, что при такой параметризации все функции $f(x)$ и $g(x)$ должны быть нормированы. Нормальное и логарифмически нормальное распределения нормированы в интервале $[-\infty, +\infty]$. Однако область определения энерговыделения уже – она простирается от 0 до верхней границы гистограммы E_{max}. Поэтому, при необходимости, функции $g(x)$ следует перенормировать в интервале $[0, E_{max}]$:

$$g(x) \to g(x) \Big/ \int_0^{E_{max}} g(x)dx$$

Как правило, верхняя граница гистограммы подбирается так, чтобы все события лежали в ее пределах. В этом случае можно считать, что $E_{max} = \infty$.

Нормировочные интегралы от нормального и логарифмически-нормального распределения выражаются аналитически через спецфункцию $\Phi(x)$:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{t^2}{2}\right) dt$$

$$\int_0^{\infty} g(x; x_0, \sigma)dx = 1 - \Phi\left(-\frac{x_0}{\sigma}\right) = \Phi\left(\frac{x_0}{\sigma}\right)$$

$$\int_0^{\infty} g_l(x; x_0, \sigma, \eta)dx = \Phi\left(\frac{1}{\sigma_0} \ln\left(1 + \frac{x_0}{\sigma}\eta\right) - \sigma_0\right)$$

В математической библиотеке ROOT функция $\Phi(x)$ доступна как `TMath::Freq()`.

К заданию прилагается два примера программы для интерактивной среды ROOT. Первый удобно использовать для подбора аналитической формы функций $f_e(x)$ и $f_\mu(x)$. В примере реализована аппроксимация энерговыделения мюонов логарифмически-нормальным распределением. Этот пример необходимо расширить, добавив возможность аппроксимации энерговыделения как мюонов, так и электронов, и подобрав подходящие формы функций. На этапе подбора аналитической формы разрешено использовать `TH1::Fit()`.

Во втором примере реализована аппроксимация энерговыделения мюонов логарифмически-нормальным распределением методом максимального правдоподобия с использованием класса `TMinuit`. Этот пример следует расширить, модифицировав функцию правдоподобия, включив в нее оба типа событий, и использовав в ней подобранные аналитические формы для $f_e(x)$ и $f_\mu(x)$. Полный список свободных параметров при минимизации функции правдоподобия должен включать число событий обоих типов и все параметры, описывающие формы распределений.

Результатом выполнения задания является картинка, на которой показаны распределения энерговыделения для электронов и мюонов, общее распределение энерговыделения и доверительный эллипс для n_e и n_μ. Для построения доверительного эллипса следует использовать метод `TMinuit::Contour(40,i,j)`, где `40` – число точек на эллипсе, а `i` и `j` – номера параметров, для которых строится эллипс. Пример картинки показан на Рисунок 26.

Основные и дополнительные вопросы

1. Сравните полученные оценки с их истинным значением (поскольку данные получены с помощью моделирования, точно известно, сколько событий какого типа). Согласуются ли полученные оценки с точным значением?
2. Как можно оценить смещение полученной оценки?
3. Какие выводы можно сделать на основе формы доверительного эллипса?

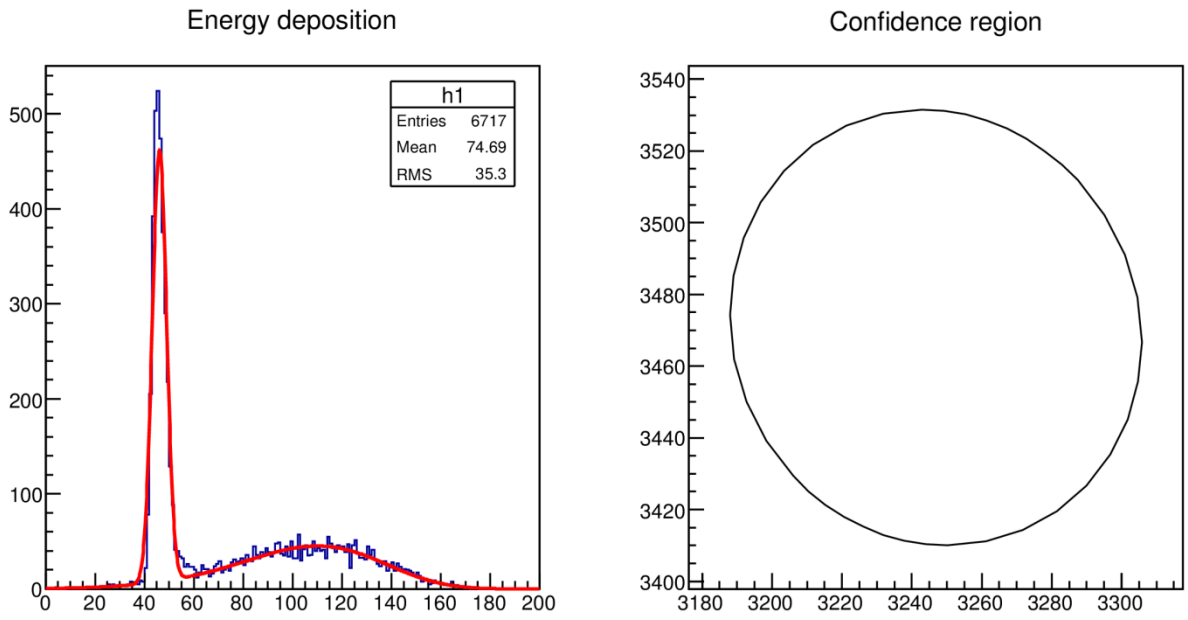


Рисунок 26. Пример подгонки энерговыделения при наличии двух типов событий

4. Пусть в результате эксперимента нужно получить не число событий каждого типа n_e и n_μ, а их сумму $(n_e + n_\mu)$ (или разность $(n_e - n_\mu)$, отношение n_e/n_μ и т.п.). Как вычислить значение суммы (отношения, разности и т.п.) и оценить ее ошибку, получив оценку n_e и n_μ с помощью метода максимального правдоподобия? Указание: необходимо учесть корреляцию между n_e и n_μ.
5. Получите оценку суммы (разности, отношения и т.п.) из вопроса 4 напрямую с помощью метода максимального правдоподобия. Совпала ли полученная оценка с расчетным значением, полученным в вопросе 4? Указание: для этого необходимо переписать функцию правдоподобия так, чтобы необходимая величина (сумма, разность, отношение и т.п.) являлась свободным параметром при минимизации.

3.4 Нейронные сети

Постановка задачи

Пусть стоит задача идентификации типа частицы, зарегистрированной детектором. В отличие от предыдущего задания, в котором было необходимо определить полное число событий каждого типа, теперь требуется идентифицировать каждое событие.

Постройте нейронную сеть, которая будет решать поставленную задачу. Используйте массив экспериментальных данных, сгенерированный в рамках задания 3.2, для обучения сети. В качестве входов сети используйте энерговыделения в каждом слое, в качестве выхода – тип частицы. В качестве сигнала используйте события с гамма-квантами, в качестве фона – события с электронами.

Результатами выполнения задания являются структура сети, распределения ответа сети для событий сигнала и фона, и кривая $\alpha - \beta$. Пример подобной сети показан на рисунке Рисунок 27.

Указания

Для реализации нейронной сети с помощью пакета ROOT удобно использовать класс `TMultiLayerPerceptron`. Для создания сети необходимо использовать конструктор:

```
TMultiLayerPerceptron *mlp = new TMultiLayerPerceptron(
                             "x,@y,@z:5:pid",
                             myt,
                             "(Entry$%2)",
                             "((Entry$+1)%2)");
```

Предполагается, что данные для обучения хранятся в дереве, указатель на которое передается вторым параметром конструктора (`myt`). Первый аргумент конструктора описывает саму сеть. Здесь `"x,@y,@z"` – это список входов сети, `5` – число нейронов в скрытом слое, `pid` – ожидаемый выход сети. Все имена входов и выходов, `x`, `y`, `z`, `pid` – это имена соответствующих листьев в дереве. Символ `@`, поставленный перед именем входа, означает, что соответствующие данные будут автоматически нормированы. Последние два аргумента описывают условия отбора для тренировочной и тестовой выборки – здесь четные события используются для тренировки, а нечетные – для контроля.

Обучение сети осуществляется с помощью вызова

```
mlp->Train(200, "text,graph,update=10,current");
```

где `200` – это количество итераций, а во втором аргументе задаются параметры обучения.

Универсальной характеристикой, по которой можно сравнивать различные классификаторы, является *кривая* $\alpha - \beta$ (или *кривая ошибок*, или *кривая ROC*). Данная кривая показывает зависимость величины ошибки первого рода α (доли событий сигнала, ошибочно идентифицированных как фон) от величины ошибки второго рода β (доли событий фона, ошибочно идентифицированных как сигнал). В системе координат (α, β) данная кривая соединяет точки с координатами $(1,0)$, что соответствует ситуации, когда все события принимаются за сигнал, и $(0,1)$, что соответствует ситуации, когда все события принимаются за фон. Чем ниже проходит кривая $\alpha - \beta$, тем лучше классификатор. Часто в качестве универсальной характеристики классификатора используют интеграл под кривой ошибок.

Для построения кривой необходимо построить распределения отклика классификатора для событий сигнала и для событий фона. Затем для различных значения порога, меняющегося от минимального значения до максимального значения классификатора, необходимо посчитать долю α событий, лежащих ниже порога в распределении отклика классификатора для событий сигнала, и долю β событий, лежащих выше порога в распределении отклика классификатора для событий фона. Получившееся множество пар значений (α, β) для различных значений порога и образует искомую кривую. Пример кривой $\alpha - \beta$ показан на рисунке 29.

Стоит отметить, что в литературе встречаются и другие определения той же самой характеристики, например зависимость $(1 - \alpha)$ от $(1 - \beta)$, или зависимость $(1 - \alpha)$ от β, и т.п. Все используемые определения тривиальным образов сводятся друг к другу.

К заданию прилагается пример, в котором генерируется простой массив данных (toy Monte Carlo) и на основе этого массива обучается нейронная сеть. Этот пример необходимо модифицировать следующим образом.

1. Изменить источник данных на дерево ROOT, сгенерированное в рамках задания 3.2.
2. Подобрать структуру сети. Для этого попробовать несколько вариантов и выбрать тот, в котором будет получена минимальная ошибка.
3. Построить кривую $\alpha - \beta$.

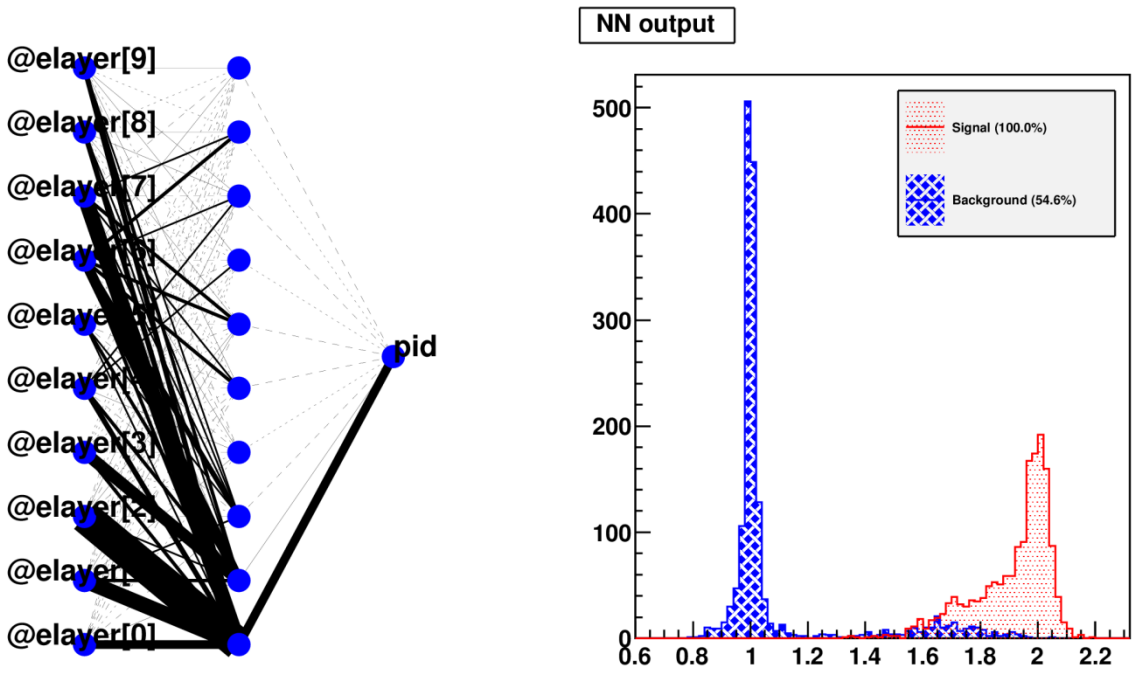


Рисунок 27. Пример классификации событий с помощью нейронной сети

Пример работы сети показан на Рисунок 27.

Основные и дополнительные вопросы

1. Нарисуйте кривые $\alpha - \beta$ для различных конфигураций сети.
2. Используйте подмножество слоев в качестве входов сети. Нарисуйте соответствующие кривые $\alpha - \beta$ для различных наборов входов и сделайте вывод о том, какие слои предоставляют больше информации о типе частицы.
3. Покажите влияние разрешения детектора на результат работы сети. Для этого модифицируйте энерговыделения в слоях, добавив к ним шум, и постройте семейство кривых $\alpha - \beta$. Сеть должна быть обучена на исходных (не модифицированных) данных.
4. Покажите влияние калибровки детектора на результат работы сети. Для этого модифицируйте энерговыделения в слоях, умножив их на масштабный коэффициент. Масштабный коэффициент должен быть выбран случайно для каждого слоя, но один и тот же коэффициент должен быть использован для всех событий. Такой подход позволяет смоделировать именно ошибку калибровки, когда существует неизвестный масштабный фактор, который не меняется от события к событию. Необходимо провести несколько раундов такого моделирования, и для каждого раунда построить кривую $\alpha - \beta$. Сеть должна быть обучена один раз на исходных (не модифицированных) данных. Ширина семейства кривых покажет влияние ошибки калибровки.

3.5 Многомерная классификация данных

Постановка задачи

В этом задании требуется решить ту же задачу, что и в предыдущем задании - необходимо идентифицировать тип частицы, зарегистрированной детектором. Однако теперь эту

задачу надо решить не с помощью нейронной сети, а с помощью других алгоритмов многомерной классификации.

Используйте массив экспериментальных данных, сгенерированный в рамках задания 3.2, для обучения алгоритмов. В качестве входных данных используйте энерговыделения в каждом слое, в качестве результата работы алгоритма – тип частицы. В качестве сигнала используйте события с гамма-квантами, в качестве фона – события с электронами.

Необходимо сравнить следующие алгоритмы:

1) отношение правдоподобий в предположении независимости входных данных;
2) подсчет числа событий в тренировочном множестве в окрестности текущего события;
3) усиленные деревья принятия решений.

Требуется реализовать эти алгоритмы, построить отклик классификатора для событий сигнала и фона и кривую $\alpha - \beta$ для каждого алгоритма, и сделать вывод, какой алгоритм лучше всего работает в данном случае. Пример результатов работы показан на рисунке 28 (отклик классификатора) и рисунке 29 (кривая $\alpha - \beta$).

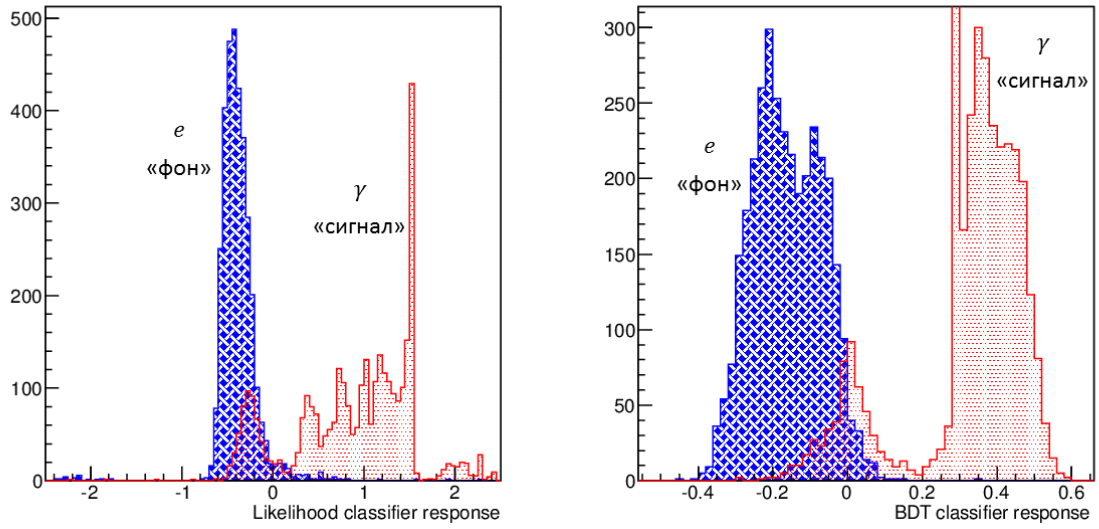


Рисунок 28. Пример отклика классификатора для событий сигнала (γ) и фона (e). Слева: в качестве классификатора используется отношение правдоподобий в предположении независимости входных данных (Likelihood); справа: классификатор построен с помощью усиленных деревьев принятия решения (BDT).

Указания

Для реализации алгоритмов следует использовать пакет TMVA, распространяющийся вместе с пакетом ROOT. В этом пакете реализованы все необходимые алгоритмы. Схема работы с TMVA описана далее.

Для обучения нескольких алгоритмов классификации необходимо произвести следующие шаги.

1. Инициализация пакета

```
gSystem->Load("libTMVA");
TMVA::Tools::Instance();
```

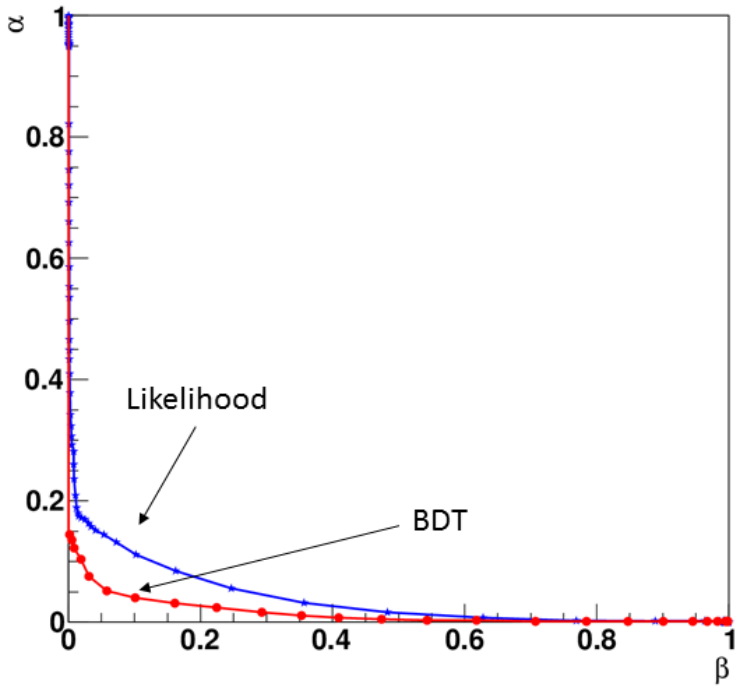


Рисунок 29. Кривые $\alpha - \beta$ для классификаторов, отклик которых показан на Рисунке 28.

2. Создание фабрики алгоритмов

```
TMVA::Factory *factory = new TMVA::Factory( "myTMVA",
                                            fout,
                                           "!V:!Silent" );
```

Весь интерфейс работы с пакетом предоставляется через фабрику. Первый аргумент – символьное имя фабрики, второй аргумент – имя файла, в котором будут сохранены результаты. В третьем аргументе задаются многочисленные параметры фабрики.

3. Определение входных данных

```
factory->SetInputTrees(myt,"pid==1","pid==2")
factory->AddVariable("el0 := elayer[0]",'F');
factory->AddVariable("el1 := elayer[1]",'F');
factory->AddVariable("el2 := elayer[2]",'F');
factory->PrepareTrainingAndTestTree();
```

Все данные для обучения лежат в дереве `myt`, которое связывается с фабрикой с помощью метода `SetInputTrees()`. Второй и третий аргумент этого метода – это условия отбора событий сигнала и фона из дерева. Далее, каждый элемент входных данных должен быть описан с помощью `AddVariable()`, при этом значение элемента (например, `elayer[0]`) – это имя соответствующего листа в дереве. После определения всех данных, необходимо провести их предварительную обработку с помощью `PrepareTrainingAndTestTree()`.

4. Создание алгоритмов классификации

```
factory->BookMethod(TMVA::Types::kMLP, "MLP",
                    "H:!V:VarTransform=N:NCycles=300:
                     HiddenLayers=N+5:TestRate=5");
```

```
factory->BookMethod(TMVA::Types::kBDT, "BDT",
                    "H:!V:NTrees=200:nEventsMin=100:
                     BoostType=AdaBoost");
```

Все алгоритмы классификации создаются с помощью единого метода `BookMethod()`. Какой именно алгоритм будет реализован определяется первым аргументом. Список основных доступных алгоритмов приведен ниже. Во втором аргументе задается символьное имя алгоритма, в третьем – параметры алгоритма. Например, в случае нейронной сети, параметр алгоритма – это количество нейронов в скрытом слое.

5. Обучение алгоритмов

```
factory->TrainAllMethods();
factory->TestAllMethods();
factory->EvaluateAllMethods();
```

Обучение и анализ всех созданных алгоритмов производится одновременно. Это позволяет удобным образом сравнивать результаты работы алгоритмов.

Перед обучением алгоритмов входные данные можно подготовить. Указать способ предварительной подготовки данных индивидуально для каждого алгоритма можно с помощью параметра "`VarTransform=`". Пакет TMVA позволяет использовать следующие алгоритмы предварительной подготовки данных:

- Нормализация данных, при которой входные данные преобразуются линейным образом так, чтобы минимальное значение соответствовало -1, а максимальное - +1. Включить нормализацию данных можно, указав `VarTransform=Norm` или `VarTransform=N`.
- Линейная декорреляция входных данных с помощью диагонализации ковариационной матрицы. В пакете реализовано два варианта подобного преобразования входных данных, которые включаются с помощью `VarTransform=Deco` (или `VarTransform=D`) в случае простой линейной декорреляции, или `VarTransform=PCA` (или `VarTransform=P`) в случае метода главных компонент.
- Рандомизация данных, при которой входные данные преобразуются нелинейным образом так, чтобы они равномерно заполняли интервал значений [0,1]. Включить подобную рандомизацию можно, указав `VarTransform=Uniform` или `VarTransform=U`.
- Рандомизация данных, при которой входные данные преобразуются нелинейным образом так, чтобы они были распределены нормальным образом. Включить подобную рандомизацию можно, указав `VarTransform=Gauss` или `VarTransform=G`.

Алгоритмы предварительной подготовки данных можно применять последовательно. Например, следующее значение параметра `VarTransform=U,P` означает сначала провести рандомизацию, затем линейную декорреляцию с помощью метода главных компонент. Выбор конкретного алгоритма предварительной подготовки данных зависит как от используемого алгоритма классификации, так и от свойств (распределения) входных данных.

Для использования обученных алгоритмов необходимо произвести следующие шаги.

1. Создать объект `TMVA::Reader`, с помощью которого осуществляется чтение параметров обученных алгоритмов:

```
gSystem->Load("libTMVA");
```

```
Reader* reader = new Reader("Color");
```

2. Определить входные данные аналогично тому, как это было сделано при обучении:

```
reader->AddVariable("elayer[0]",&(el[0]));
reader->AddVariable("elayer[1]",&(el[1]));
reader->AddVariable("elayer[2]",&(el[2]));
```

Первым аргументом метода `AddVariable()` является имя переменной (которое должно совпадать с именем, использованном при обучении). Вторым аргументом является указатель (адрес ячейки памяти), в которой хранится значение переменной. При дальнейшей работе с классификатором, сначала указанные ячейки памяти заполняются значениями входных данных для очередного события (например, считываются из дерева ROOT), а потом вычисляется значение классификатора.

3. Определить заранее обученные алгоритмы классификации

```
reader->BookMVA("Likelihood",
                "weights/myTMVA_Likelihood.weights.xml");
```

После обучения, все параметры обученного классификатора сохраняются в XML файле в поддиректории `weights`. Имя файла определяется общим префиксом, указанном во время обучения при создании фабрики `TMVA::Factory` (в этом примере "`myTMVA`"), и именем алгоритма, указанном в методе `TMVA::BookMethod()` (в этом примере "`Likelihood`").

4. Вычислить значения классификатора

```
reader->EvaluateMVA("Likelihood");
```

В TMVA доступны следующие алгоритмы (и ряд других):

- TMVA::Types::kLikelihood – отношение правдоподобий в предположении независимости входных данных.

  В этом алгоритме явно предполагается, что во входных данных отсутствует корреляция. Обучение алгоритма сводится к тому, что для каждого из входных данных строятся одномерные распределения, отдельно для сигнала и фона, которые затем сглаживаются либо с помощью сплайнов, либо с помощью метода ядерного сглаживания (kernel functions). Далее строится многомерная функция правдоподобия для сигнала и фона как произведение полученных плотностей распределения входных данных: $L_{S,B}(x_1, x_2, \dots) = \prod f(x_i^{S,B})$, где S и B обозначают сигнал и фон, соответственно, x_i – входные данные, $f(x)$ – одномерная плотность распределения соответствующего данного. Классификатор строится как отношение функций правдоподобия сигнала и фона, в соответствии с леммой Неймана-Пирсона:

$$t(x_1, x_2, \dots) = \frac{L_S(x_1, x_2, \dots)}{L_S(x_1, x_2, \dots) + L_B(x_1, x_2, \dots)}.$$

  Часто значения такого классификатора оказываются слишком пикованы в районе нуля и единицы, что неудобно при практическом применении. Для того, чтобы избежать этой проблемы, к значению классификатора можно применить монотонное преобразование

$$t \to \frac{1}{15} \ln\left(\frac{1}{t} - 1\right)$$

которое «расширяет» пики в нуле и единице. Такое преобразование включается с помощью параметра `TransformOutput=True`.

Пример определения этого алгоритма:

```
factory->BookMethod(TMVA::Types::kLikelihood, "Likelihood",
                    "H:!V:VarTransform=N,D:
                   PDFInterpol=Spline2:TransformOutput=True");
```

В данном примере используется сглаживание одномерных плотностей вероятности с помощью сплайна второго порядка.

- TMVA::Types::kPDERS – подсчет числа событий в окрестности текущего события (PDE-RS).

  В данном алгоритме классификатор строится тоже как отношение правдоподобий, однако вместо значения функции правдоподобия используется число событий сигнала и фона в заданной окрестности. Если окрестность достаточно мала, то число событий пропорционально значению функции правдоподобия. Обучение для этого алгоритма как таковое не требуется. При использовании классификатора подсчитывается число событий сигнала и фона, попадающих в заданную гиперсферу вокруг классифицируемого события, и вычисляется

$$t(x_1, x_2, \dots) = \frac{1}{1 + (n_B(V)/N_B)/(n_S(V)/N_S)},$$

  где V – заданная окрестность вокруг точки $(x_1, x_2, \dots)$, $n_{S,B}(V)$ – число событий сигнала и фона, соответственно, внутри этой окрестности, $N_{S,B}$ – полное число событий сигнала и фона в тренировочном массиве.

  Пример определения этого алгоритма:

```
factory->BookMethod(TMVA::Types::kPDERS, "PDERS",
                    "H:!V:
                     VolumeRangeMode=RMS:
                     KernelEstimator=Gauss");
```

  В данном примере используется окрестность с размытыми (гауссовыми) границами, размеры которой по каждой оси пропорциональны дисперсии соответствующего входного данного.

- TMVA::Types::kKNN – подсчет доли событий сигнала среди k ближайших соседей текущего события (PDE-kNN).
  Данный алгоритм похож на предыдущий с тем отличием, что вместо окрестности предопределенного размера используется такая окрестность, в которую попадает k событий. Такая реализация классификатора гораздо эффективнее – на этапе обучения составляется дерево, в котором для каждого события в тренировочном наборе указаны k его ближайших соседей, поэтому на этапе использования нет необходимости в переборе всего тренировочного массива.

  Пример определения этого алгоритма:

```
factory->BookMethod(TMVA::Types::kKNN, "KNN",
                    "H:!V:
                     nkNN=50");
```

  В данном примере используется окрестность, в которую попадает 50 событий.

- TMVA::Types::kFisher – линейный дискриминатор Фишера.

  Простой линейный дискриминатор, представляет собой такую линейную сумму входных данных, которая максимальным образом отличается для сигнала и фона.

  Пример определения этого алгоритма:

```
factory->BookMethod(TMVA::Types::kFisher, "Fisher",
                    "H:!V");
```

- TMVA::Types::kBDT – усиленные (boosted) деревья принятия решений (decision trees).

  Мощный нелинейный классификатор, не уступающий более популярным нейронным сетям. Алгоритм основан на автоматическом выделении областей пространства, в которых лежат в основном события сигнала или фона. Границы областей задаются в виде последовательности простых условий вида $x_i < a$ или $x_i > b$. Совокупность таких условий, которые определяют область сигнала, можно представить в виде дерева условий (дерева принятия решения). Одно дерево представляет собой статистически неустойчивый классификатор, который слишком сильно зависит от флуктуаций в тренировочном наборе. Поэтому классификатор строится на основе большого множества деревьев (леса), при этом при построении каждого следующего дерева усиливается роль событий, неправильно классифицированных с помощью ранее построенных деревьев. Такая процедура усиления классификатора называется бустингом (boosting).

  Пример определения этого алгоритма:

```
factory->BookMethod(TMVA::Types::kBDT, "BDT",
                    "H:!V:
                    NTrees=800:MaxDepth=5:
                    SeparationType=GiniIndex:
                    BoostType=AdaBoost");
```

  В данном примере определен лес, состоящий из 800 деревьев принятия решений. Глубина каждого дерева не превышает 5 уровней. При формировании деревьев автоматический подбор границ ячеек производится на основе индекса Гини (равного $p \cdot (1 - p)$, где p – доля событий сигнала в ячейке). Построение леса осуществляется с помощью алгоритма AdaBoost (одного из распространенных алгоритмов бустинга).

- TMVA::Types::kMLP – нейронная сеть (многослойный перцептрон).

  В качестве классификатора здесь используется нейронная сеть (многослойный перцептрон), обученная выдавать значение 0 для событий фона и значение 1 для событий сигнала. В пакете TMVA используется другая реализация нейронной сети, чем в классе `TMultiLayerPerceptron` (см. задание 3.4). Реализация TMVA предоставляет больше возможностей: можно использовать различные функции активации нейронов (включая радиальную); можно использовать различные алгоритмы обучения (классический алгоритм обратного распространения ошибок, алгоритм BFGS, генетические алгоритмы); и т.п.

  Пример определения этого алгоритма:

```
factory->BookMethod(TMVA::Types::kMLP, "MLP",
                    "H:!V:
                    HiddenLayers=N+5,3:NeutronType=sigmoid:
                    TrainingMethod=BP:BPMode=batch");
```

В данном примере определена сеть с двумя скрытыми слоями (в первом слое на 5 нейронов больше, чем входных данных, во втором слое 3 нейрона), в качестве функции активации используется сигмоида, обучение проводится с помощью алгоритма обратного распространения с пакетной коррекцией весов.

Выбор конкретного алгоритма классификации зависит в первую очередь от структуры самих данных. Как правило, оценить применимость различных классификаторов в каждом конкретном случае можно только методом проб и ошибок. Если в данных присутствует только линейная корреляция, то простые классификаторы, такие как отношение правдоподобий или линейный дискриминатор, могут оказаться оптимальными. Однако при наличии больших нелинейных корреляций, свойства этих классификаторов значительно ухудшаются. Часто на выбор могут повлиять практические соображения. Например, если требуется вычислять классификатор на ограниченных ресурсах, то алгоритмы PDE-RS или kNN могут оказаться плохим выбором, так как они требуют большого объема памяти (фактически, их обучение сводится к запоминанию всего тренировочного массива, а вычисление соответствующего классификатора – к поиску в этом массиве). В качестве еще одного примера можно заметить, что, хотя классификатор на основе нейронной сети теоретически обладает оптимальными характеристиками, при условии правильного выбора архитектуры сети и наличии достаточного количества нейронов, на практике менее оптимальный классификатор на основе усиленных деревьев принятия решения показывает лучшие результаты. Связано это со сложностью обучения нейронной сети.

К заданию прилагается пример, в котором обучается один алгоритм и используется ограниченное число входных данных. Этот пример необходимо модифицировать, дополнив список входных данных и реализовав необходимые алгоритмы.

Основные и дополнительные вопросы

1. Оцените, улучшает ли предварительная подготовка данных с помощью метода главных компонент характеристики выбранного алгоритма классификации.
2. С какими весами следует сложить энерговыделения в слоях калориметра, чтобы наилучшим образом разделить электроны и гамма-кванты?
3. Покажите влияние разрешения детектора на результат работы сети. Для этого модифицируйте энерговыделения в слоях, добавив к ним шум, и постройте семейство кривых $\alpha - \beta$. Выбранные алгоритмы должны быть обучены на исходных (не модифицированных) данных. Характеристики какого алгоритма ухудшаются наибольшим образом при добавлении шума? Восстанавливаются ли исходные характеристики выбранных алгоритмов, если обучить их на зашумленных данных?

Литература

1. Rene Brun and Fons Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.
2. Официальный веб-сайт проекта ROOT. http://root.cern.ch/drupal/
3. ROOT Users Guide. http://root.cern.ch/drupal/content/users-guide
4. ROOT Reference Guide. http://root.cern.ch/drupal/content/reference-guide
5. ROOT Primer. http://root.cern.ch/drupal/content/users-guide#primer
6. ROOT Tutorials. http://root.cern.ch/drupal/content/tutorials
7. Allison, J. et al., *Geant4 developments and applications* // IEEE Transactions on Nuclear Science, 53 (2006) 270 — 278.
8. Официальный веб-сайт проекта Geant4. http://geant4.cern.ch/
9. Geant4 User's Guide: For Application Developers. http://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/ForApplicationDeveloper/html/index.html
10. Geant4 User's Guide: For Toolkit Developers. http://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/ForToolkitDeveloper/html/index.html
11. Geant4 Physics Reference Manual. http://geant4.web.cern.ch/geant4/UserDocumentation/UsersGuides/PhysicsReferenceManual/fo/PhysicsReferenceManual.pdf
12. TMVA User’s Guide. http://tmva.sourceforge.net/docu/TMVAUsersGuide.pdf
13. Логашенко И.Б. «Методы анализа экспериментальных данных». Электронный курс лекций.